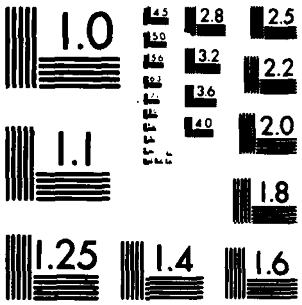
AD-A107 249 IBM FEDERAL SYSTEMS DIV GAITHERSBURG MD F/G 9/2
PROGRAMMING SUPPORT LIBRARY (PSL). USERS MANUAL.(U)
MAY 78 F30602-77-C-0249
UNCLASSIFIED DOD/DF-81/016A NL

1 of 3



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

a

.NITED

Programming Support Library (PSL)

Users Manual

IBM

50272-101

REPORT DOCUMENTATION PAGE

1. REPORT NO. 19 18 DOD/DF 81/Ø16a

2.

3. Recipient's Accession No. AD-A1Ø7 249

4. Title and Subtitle

6 PROGRAMMING SUPPORT LIBRARY (PSL), Users Manual,

5. Report Date 11 May 1978

6.

7. Author(s)

8. Performing Organization Rept. No. 9 Final rept.

9. Performing Organization Name and Address

Federal Systems Division
International Business Machines Corporation
Gaithersburg, Maryland

10. Project/Task/Work Unit No.

11. Contract(C) or Grant(G) No.

15 (C) F3Ø6Ø2-77-C-Ø249

(G)

12. Sponsoring Organization Name and Address

Rome Air Development Center
RADC/COEE
Griffiss Air Force Base, NY 13441

13. Type of Report & Period Covered

14. 12 272

15. Supplementary Notes

21 For magnetic tape, see AD-A1Ø7248.
See Also AD-A1Ø7 25Ø.

The source agency has restricted sales of this item to Federal, state and local governments.

16. Abstract (Limit: 200 words)

The Programming Support Library (PSL) is a software system which provides the tools to organize, implement, and control computer program development. This involves the support of the actual programming process and also the support of the management process. The PSL is designed to support Top Down Design and Structured Programming (TDDSP).

The Users Manual presents a summary of the PSL system, a description of the PSL processing, and a guide for users of the system. This document is prepared for those who need an overall understanding of the system and for those who need more detailed technical information on the use of the system.

17. Document Analysis a. Descriptors

Software Configuration Control
Structured Programming Support Tool
Software Development Support

b. Identifiers/Open-Ended Terms

c. COSATI Field/Group

174950

18. Availability Statement:	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages
	20. Security Class (This Page) UNCLASSIFIED	22. Price

223

See Instructions on Reverse

DMA
Programming Support Library (PSL)
Users Manual

(FINAL)

Submitted to:

Rome Air Development Center
Griffiss AFB, New York

MAY 1978

Under Contract F30602-77-C-0249

Federal Systems Division
INTERNATIONAL BUSINESS MACHINES CORPORATION
Gaithersburg, Maryland

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF TABLES

ABSTRACT

This Users Manual presents a summary of the Defense Mapping Agency (DMA) Programming Support Library (PSL) system, a description of the PSL processing, and a guide for users of the system. This document is prepared for those who need an overall understanding of the system and for those who meed more detailed technical information on the use of the system.

SECTION 1. GENERAL DESCRIPTION

1.1 Purpose

The Defense Mapping Agency (DMA) Programming Support Library (PSL) is a software system which provides the tools to organize, implement, and control computer program development. The system is designed specifically to support top-down development and structured programming, but non-structured programs can also be accommodated. This Users Manual provides the documentation required to initialize and develop a programming project under the PSL system in a batch mode on Sperry Univac 1100 series hardware.

The Users Manual is organized such that the text becomes increasingly detailed and specific as the reader progresses. The general progression is as follows:

a. Section 1 - General Description. This section is devoted to the general concept of the new programming techniques and summarizes relevant reference material.

b. Section 2 - System Summary. This section describes the general capabilities of the PSL system and the concept of a programming library.

c. Section 3 - Technical Operation. This section describes the specific capabilities of the system, in the following order:

(1) Subsection 3.1 - Overview of Input Requirements. This subsection describes the basic capabilit es of each Function directive which may be processed by the PSL system. The overview is intended to give the user a high-level introduction to the use of the system before specific details of format are introduced. The Function directives are discussed in the order in which they would logically occur in actual use.

(2) Subsection 3.2 - Composition Guides. This subsection covers many miscellaneous technical items which have general application to more than one PSL Function directive.

(3) Subsection 3.3 - Conventions and Glossary. These items establish standards of format and vocabulary to be used in subsection 3.4.

(4) Subsection 3.4 - Input Format of Function Cards. This subsection discusses each PSL Function directive in detail. It is intended to be used as reference material and is arranged in alphabetical order by Function.

1.2 Project Background

There have been, from the beginning of programming activities, certain principles that good programmers have identified and practiced in one way or another. These include: developing system designs from a gross level to more and more detail until the detail of a computer operation is reached; dividing a system into modules in such a way that minimal interaction takes place through module interfaces; and using meaningful terms in the coding process.

Two key principles, which are new in their application to programming, will play a major role in the implementation and exploitation of these ideas.

The first key technical principle is that the control logic of any program can be designed and coded in a highly structured way. In fact, arbitrarily large and complex programs can be represented by iterating and nesting a small number of basic and standard control logic structures.

A practical application of this first principle is the writing of structured programs, i.e., programs in which the branching control logic can be defined entirely in terms of DO loops, IF-THEN-ELSE, and sequences. The resulting code can be read strictly from top to botton, typographically, and is much more easily understood. It takes more skill and analysis to write such code, but its debugging and maintenance are greatly simplified. Even more importantly, such structured programming can increase a single programmer's span of detailed control and productivity by a large amount.

The second key technical principle is that programs can be coded on a schedule that requires no simultaneous interface assumptions. Programs can be coded in such a way that every interface is defined initially and uniquely in the coding process itself, and referred to thereafter only in its previously coded form.

In practical application, this second principle leads to top-down development, where code is generated in an execution precedence form. In this case, programmers write job control code first, then linkage code, and then source code. The opposite (and typical implementation procedure) is bottom-up programming, where source modules are written and unit tested first, and later integrated into subsystems and, finally, systems. This latter integration process, in fact, tests the proposed solutions of simultaneous interface problems generated by lower-level programming; and the problems of system integration and debugging arise from imperfections in these proposed solutions. Top-down programming circumvents the integration problem by the coding sequence itself.

1.2.1 Structured Programming

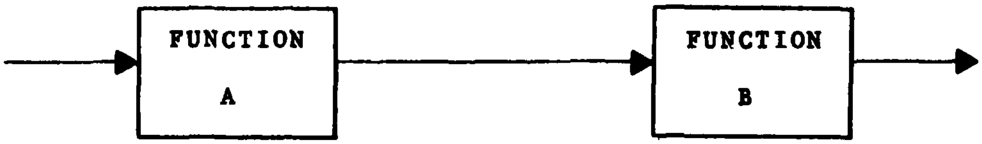
Structured programming is based on a mathematically proven "Structure Theorem" which states that any program with one entry and one exit is equivalent to a program that contains combinations of, at the most, the following logic structures:

a. Sequence of statements

b. IF-THEN-ELSE

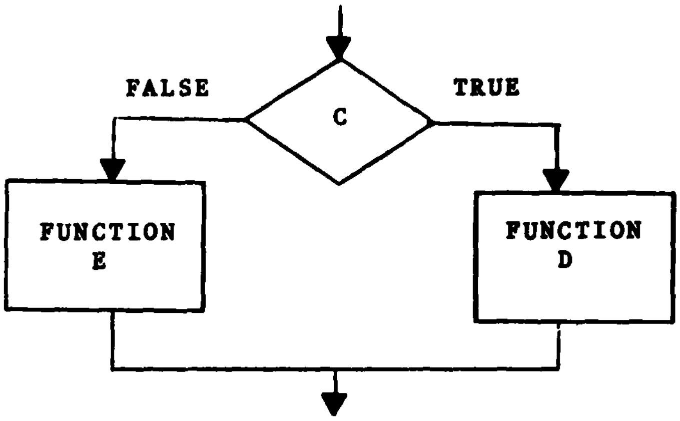
c. DO-WHILE

The Sequence logic structure is represented by the following diagram:



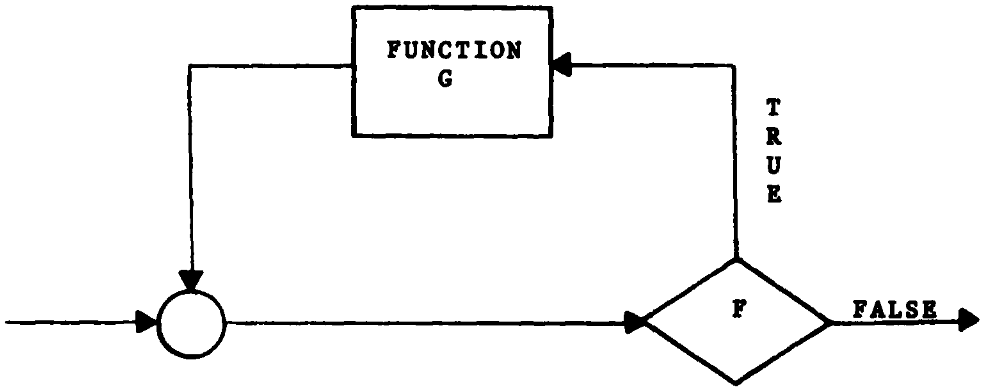
This logic structure is the simplest and indicates that function A is to be performed first, function B is to be performed next, and then processing is to continue.

The second of these basic structures, the IF-THEN-ELSE figure, can be represented by the following diagram:



In this basic structure, the flow of control is governed by the condition C. If condition C is true, function D is performed. In either case, control returns to a common node and proceeds from that node.

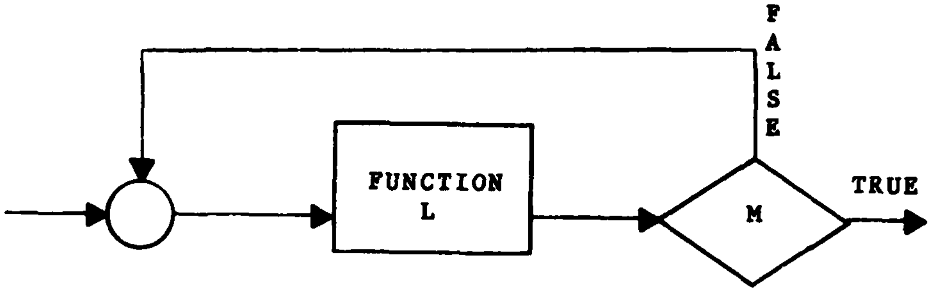
The third basic structure, the DO-WHILE figure, is represented as follows:



In this basic logic structure, the flow of control again is governed by a condition. Function G is performed while condition F is true. When condition F is no longer true, the flow of control falls through and processing continues.

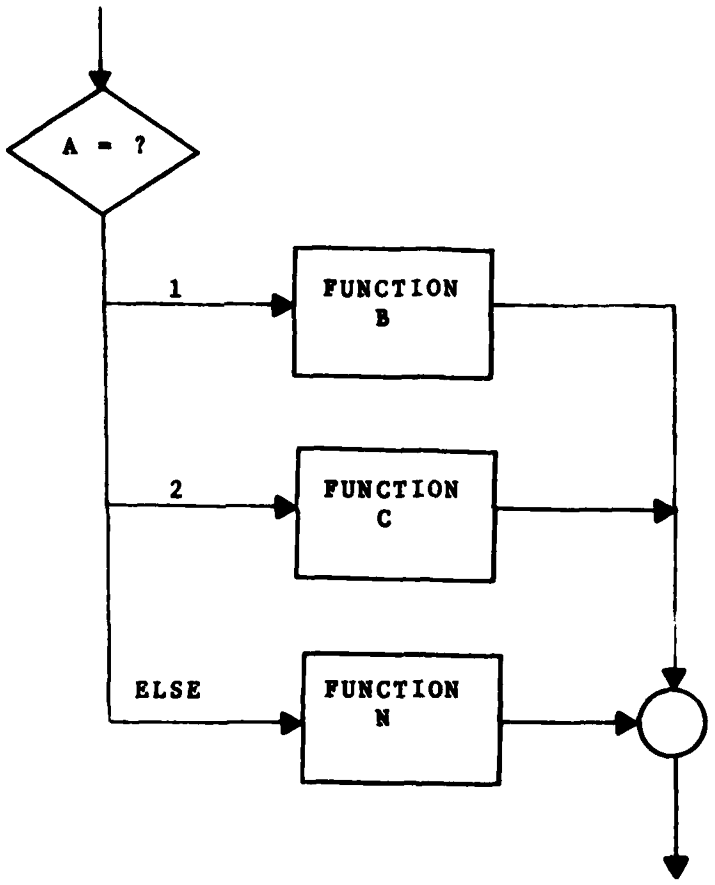
The three figures discussed above (Sequence, IF-THEN-ELSE, and DO-WHILE) are the basic figures of the structured programming theory. Other figures which one might implement are based on combinations of these figures.

A standard extension of the three basic figures is the DO-UNTIL figure, which is logically equivalent to a Sequence figure followed by a DO-WHILE figure. The DO-UNTIL figure is represented as follows:



In this structure, function L is always performed at least once. Function L will be repeated until condition M is true. Note that the flow of control falls through on a true condition, in contrast to a DO-WHILE figure, where the flow falls through on a false condition.

Another extension to the basic figures is the CASE logic structure. Although the CASE figure can be represented by a series of IF-THEN-ELSE figures, it is often desirable to represent this situation by the following structure:



In the CASE logic structure, the flow of control is governed by the value of A. If A is correctly evaluated as 1, then function B is performed. If A is 2, function C is performed. For all other evaluations of A, function N is performed. In every case, control returns to a common node and proceeds from that node.

The logic figures described above are useful at all phases of development, including the design phase. Using the figures with non-programming text, a programmer can provide procedural definitions at any level of completeness in a form referred to as Program Design Language (PDL). As the design materializes, the PDL text can evolve, in top-down fashion, into the implementation (programming) language of choice.

1.2.2 Preprccessors

The implementation of the basic logic figures will depend to some degree on the programming language being used. Once the basic figures are implemented, however, the use of these figures will enhance the ability of programmers to understand and verify each other's programs.

The logic structures used in the American National Standard (ANS) COBOL language, for example, do not correspond to those of structured programming. A specific case is the use of a dependent IF clause which is limited to logic which only progresses downward. (A dependent IF clause cannot be terminated without terminating the higher-level IF (or ELSE) code on which it is dependent.) In addition, the use of a PERFORM statement to accomplish the logic of a DO-WHILE structure results in code which is dispersed over many pages and introduces confusion to a programmer reading the code.

In order to support these structures more precisely, preprocessors (also referred to as precompilers) are used to read the structured source code and to generate new source code which is subsequently compiled. The new code is not necessarily in structured format, but the programmer does not work with this code. It is an intermediate product between his source code and his object code.

A General Preprocessor has been added to handle unstructured source code for the following languages: ASM, COBOL, FORTRAN, and JOVIAL. This General Preprocessor is an alternative to the method in section 3.2.3 for dealing with unstructured source code. A user description of the General Preprocessor is given in Appendix I.

The preprocessors also support the segmentation of source code into page-size segments called units by allowing the programmer to INCLUDE additional units into his input stream.

1.2.3 Top-Down Development

Top-down program development requires that programming proceed from the control statements downward until all functional units are developed and integrated. The process provides continuous integration of the system parts as they are programmed.

Conceptually, top-down development proceeds downward from a single starting point, while conventional implementation proceeds upwards from as many starting points as there are modules in the design. The single starting point does not imply that implementation must proceed down the hierarchy in parallel. Some branches may be developed earlier than others in order to obtain some initial operational capabilities.

Each program or module within the system should also be developed from the top downward. The same rules and benefits apply to single programs and to complete systems. A basic rule for software development using the top-down approach is that code being developed depends only on operational code.

A separate system integration period is eliminated by top-down development, since the system parts are continuously integrated. In effect, the higher-level code becomes the driver program for the most critical units are also the most tested.

Since the higher-level units will normally invoke lower-level units, dummy code (stubs) are substituted temporarily for the lower-level units to preserve the designed interface. The lower-level units are expanded and tested after the higher-level units are tested and integrated. This procedure is iterated downward, substituting actual program units at each successively lower level until the entire system has been integrated and tested.

Top-down development provides the ability to evolve a product which is always operable, modular, and available for testing. The quality of a system which is produced using the top-down approach is increased through the earlier detection and elimination of design problems and coding errors.

1.2.4 Development Support Library

Top-down programs which are written to be highly readable, and whose major structural characteristics are given in hierarchial top-down form, can be read sequentially. The programs are written in small segments (units), usually under a page in length, such that each segment can be literally read from top to bottom, with complete assurance that all control paths are visible in the segment under consideration.

There are two requirements through which one can achieve this goal. The first requirement is structured programming i.e., the formulation of programs in terms of a few standard and basic control structures, such as IF-THEN-ELSE statements, DO loops, and CASE statements, with no arbitrary jumps between these standard structures. The second requirement is library support, so that the segments themselves can be stored under symbolic names in a library and substituted at any point in a program by a special statement (INCLUDE). A key aspect of any INCLUDEd unit is that its control should enter at the top and exit at the bottom, and have no other means of entry to, or exit from, other parts of the program. Thus, when reading an INCLUDE statement, at any point, the reader can be assured that control will pass through that INCLUDEd unit and not otherwise affect the control logic on the page he is reading.

A unit of code is usually limited to 50 lines or less. Each unit of code (whose internal operations may be any combination of the basic logic structures) must itself represent one of the basic logic structures. Thus, each code unit becomes a logical entity to be analyzed, coded, and read at one time.

In order to satisfy the unit entry/exit requirement, one need only be sure to include all matching control logic statements on a page. For example, the ENDDO to any DO, and the ENDIF to any IF-THEN-ELSE, should be put in the same unit.

The end result is a program, of any size whatsoever, which has been organized into a set of named member units, each of which can be read from top to bottom without any side effects in control logic, other than what is on that particular page. A programmer can access any level of information about the program, from highly summarized, at the upper-level segments, to completely detailed, in the lower level.

A development support library is required to support such a programming effort and to provide project visibility. The library system should be designed to support a top-down structured environment. The principal objective of the library is to provide constantly up-to-date representations of the program segments and test data, in both machine-readable and hard-copy forms. In addition, the library provides the capability to obtain development status information for management control. The library is designed to maintain the current status of such items as test data, control data, source data, object modules, and load modules.

A librarian may be used to maintain the development support library. The librarian, through predefined procedures, maintains the programs and test data on disk storage. The information on disk is referred to as the internal library. The hard-copy, or external library, is maintained with a standard set of office procedures.

A development project normally contains an operational library of code that has been tested, and one or more development libraries for newly developed code. At any point in time, the operational library constitutes the current operational system. Configuration control is ensured by requiring that an update to the operational library be conditioned on proof of successful testing in a development library. This minimizes the likelihood of errors entering the system. Verification procedures are reviewed for an update to the operational library. The development support library thus provides the necessary control to develop a system in a top-down manner.

The procedures associated with the library are designed to support the practices of top-down structured programming. For example, in structured programming, strict attention is paid to the indentation of the logic structures on the printed page so that logical relationships in the coding correspond to physical position on the listing. A pictorial representation of the logic is thus expressed by the indentation. The library procedure, which lists source code, automatically sets up the indentation. This method not only eliminates work for the programmer, but also assures the code reader that the logic structure actually follows the dependencies shown in the listing and is not the result of a mistaken assumption by the programmer.

In addition, the library input procedure support the practice of unit-level development by providing a capability for automatic stub generation. Testing and integration will start with the highest-level module as soon as it is coded. Since this module will normally invoke or include lower-level segments, code must exist for the next-lower-level segment. This code, in the form of a source stub, can be provided automatically by the library system. These stubs will later be expanded into full functional units, which in turn require lower-level segments.

The library provides a central source of status data to measure the development process. Parameters, such as the number of updates performed, the number of units involved, the number of source lines present, and the number of stubs remaining, can be used to develop management reports, and to highlight problem areas which need attention.

Since the developing system is undergoing continuous integration, the system status is accurately reflected through the contents of the library. Completness is measured objectively in terms of how much of the system is operational. The completed code can be reviewed to verify status and appraise the quality of the software product.

1.2.5 Management Data Collection and Reporting (MDCR)

The purpose of the Management Data Collection and Reporting facility is to support the collecting and reporting of management data for a PSL project. Although designed as an integral part of the PSL, this facility is optional and can be selectively omitted at the project level with no effect on other functions performed for that project.

The MDCR facility involves the collection and storage of data related to program development and maintenance and the generation of management reports containing the data and/or summaries of data. The data collected is stored in a project's Management Data Section. A Management Data Section is created and maintained for each project for which management data is collected.

A capability is provided to define elements of the project for which management data is to be collected. The data items to be collected can be specified and their arrangement in subsequent management reports defined. Manual data may be input for collection and summarization with data that is automatically supplied from Unit Accounting Records. Collected data may be archived as required, and both current and archived data can be printed in defined report formats. Special capabilities are provided to annotate management reports when data item values are outside allowable ranges and to support the execution of user routines to satisfy unique data collection requirements.

1.2.6 Summary

The techniques described in the preceding paragraphs represent a disciplined approach to application development, which has as its goal, improved program quality, maintainability, and manageability. The PSL system provides the tools necessary to fully implement these new programming techniques.

The facilities described in this manual constitute the implementation of the PSL system for Rome Air Development Center under Air Force Contract F30602-77-C-0749. The requirements for the PSL system were developed as part of the Structured Programming Series of reports, RADC-TR-74-300, Volume I through XV, March 1975, under Air Force Contract F30602-74-C-0186. The following volumes have particular application to the PSL system development:

a.	Volume II	-	Precompiler Specifications
b.	Volume III	-	Precompiler Program Documentation
c.	Volume V	-	Programming Support Library Functional Requirements
d.	Volume VI	-	Programming Support Library Program Specifications
e.	Volume IX	-	Management Data Collection and Reporting

Precompilers are described in the Structured Programming Series, Volume III. The syntax acceptable to these precompilers has been modified under the PSL system to additionally allow the INCLUDE statement. PSL supports precompilers for COBOL, JOVIAL and FORTRAN. In addition, the General Preprocessor allows the INCLUDE statement to be used in unstructured ASM, COBOL, FORTRAN and JOVIAL. See Appendix I.

The COBOL source code, which is called Structured COBOL (SCOBOL) in this manual, accepts the following special sets of statements:

a. INCLUDE

b. CASENTRY
CASE
ELSECASE
ENDCASE

c. DO WHILE
   ENDDO

d. DO UNTIL
   ENDDO

e. IF
   ELSE
   ENDIF

The use of the INCLUDE statement is described in Section 3.2.8 of this Users Manual. The user should refer to Appendix G for more complete documentation on the use of Structured COBOL.

The Structured FORTRAN precompiler is discussed in Appendix F of this manual. The syntax acceptable to this precompiler allows the use of standard structured programming constructs. In addition to the INCLUDE, CASENTRY, DOWHILE, DOUNTIL, and IF statement sets listed above, the FORTRAN precompiler's structured programming language, SPFORT, provides for one additional statement set.

```
INVOKE
BLOCK
ENDBLOCK
```

The statement set performs a similar function as that provided by the INCLUDE capability. Use of the INCLUDE statement is described in Section 3.2.8. Use of the INVOKE, BLOCK, ENDBLOCK construct is described in Appendix F.

The Structured JOVIAL precompiler is discussed in Appendix H of this manual. The syntax acceptable to this precompiler contains the following special sets of statements:

a. IF
   ELSE (optional)
   ENDIF

b. DO (WHILE or UNTIL)
   ENDDO

c. CASENTRY
   CASE (at least one must be present)
   ELSECASE (optional)
   ENDCASE

d. INCLUDE

SECTION 2. SYSTEM SUMMARY

2.1 System Application

The DMA PSL is a comprehensive system software package which supports the growth and maintenance of structured programming projects in a top-down development environment. The system provides:

a. A framework for the organization of a project

b. Simple functional statements to interface between the programmer and the machine

c. Structural and statistical reports for control of the development process and for communication between programmers.

The DMA PSL system provides special structured programming support for the Structured COBOL, JOVIAL and FORTRAN languages However, unstructured programs may also be stored and maintained under the system.

2.2 System Operation

The PSL system in the batch mode is invoked by an @ADD, @END, @ADD sequence and is directed to perform specific functions by PSL Function cards. The general functional capabilities available under the PSL are:

a. Initialize a project

b. Create sections in a library

c. Add, change, move, replace, or purge a unit of code

d. Print, punch, or write (to tape) a unit of code

e. Print an index listing

f. Print the top-down structure of a program

g. Compile, link, execute

h. Delete a section, a library, or a project

i. Backup a project, library, or section.

j. Restore a project, library, section, or unit.

k. Collect and print management data.

l. Print text material

m. Print by author or character string

The PSL Functions are described in detail in Section 3. The capitalized word "function" is used to refer to one of the 28 PSL operations which is invoked with a PSL card. A general system flow chart is shown in Figure 2-01.

2.3 System Configuration

The PSL system operates on an Sperry Univac 1100 series computer under the Exec 8 Operating System using a standard DMA software and hardware configuration. Libraries are maintained on direct access storage devices. The system utilizes the standard card reader and printer, one tape drive, and 40K words of core.

2.4 System Organization

The PSL system is designed to support the program development and maintenance effort of an entire organization. In a large organization, the system must support many persons working on different programming projects which may be completely unrelated. The PSL provides means of maintaining control over all the data related to each project.

One means of control is the convention used for identifying and organizing the data (source code, compiled modules, test data, etc.) stored under a project. This convention subdivides the data, beginning with the programming project level and proceeding down to single logical units of data. The hierarchical structure of a project is shown in Figure 2-02.

There are four levels of data identification under the PSL. The highest level of identification is a project. This corresponds to a major programming operation and consists of all of the data related to that operation.

The next level of identification is a library. Each project is composed of one or more libraries. Multiple libraries can be used effectively to provide version control over the programming process and to support top-down program development and integration. Any number of libraries may exist under a given project.

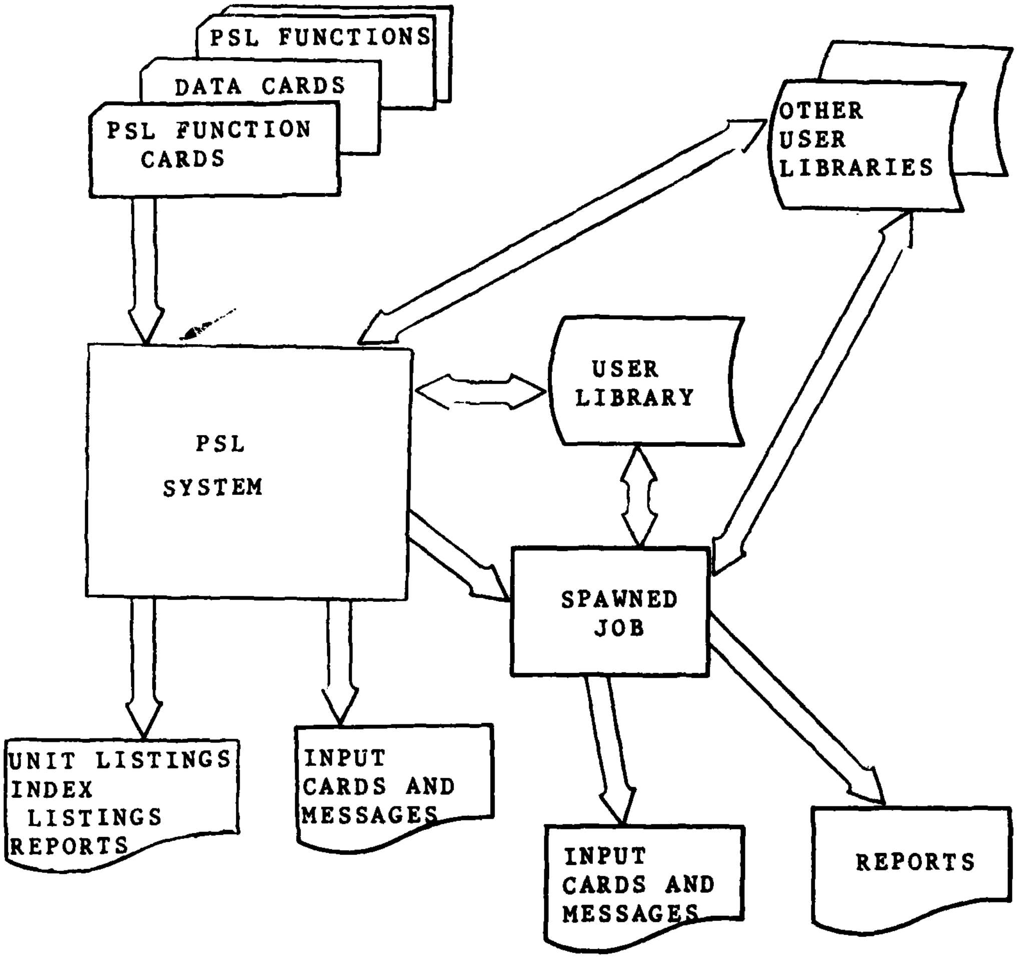
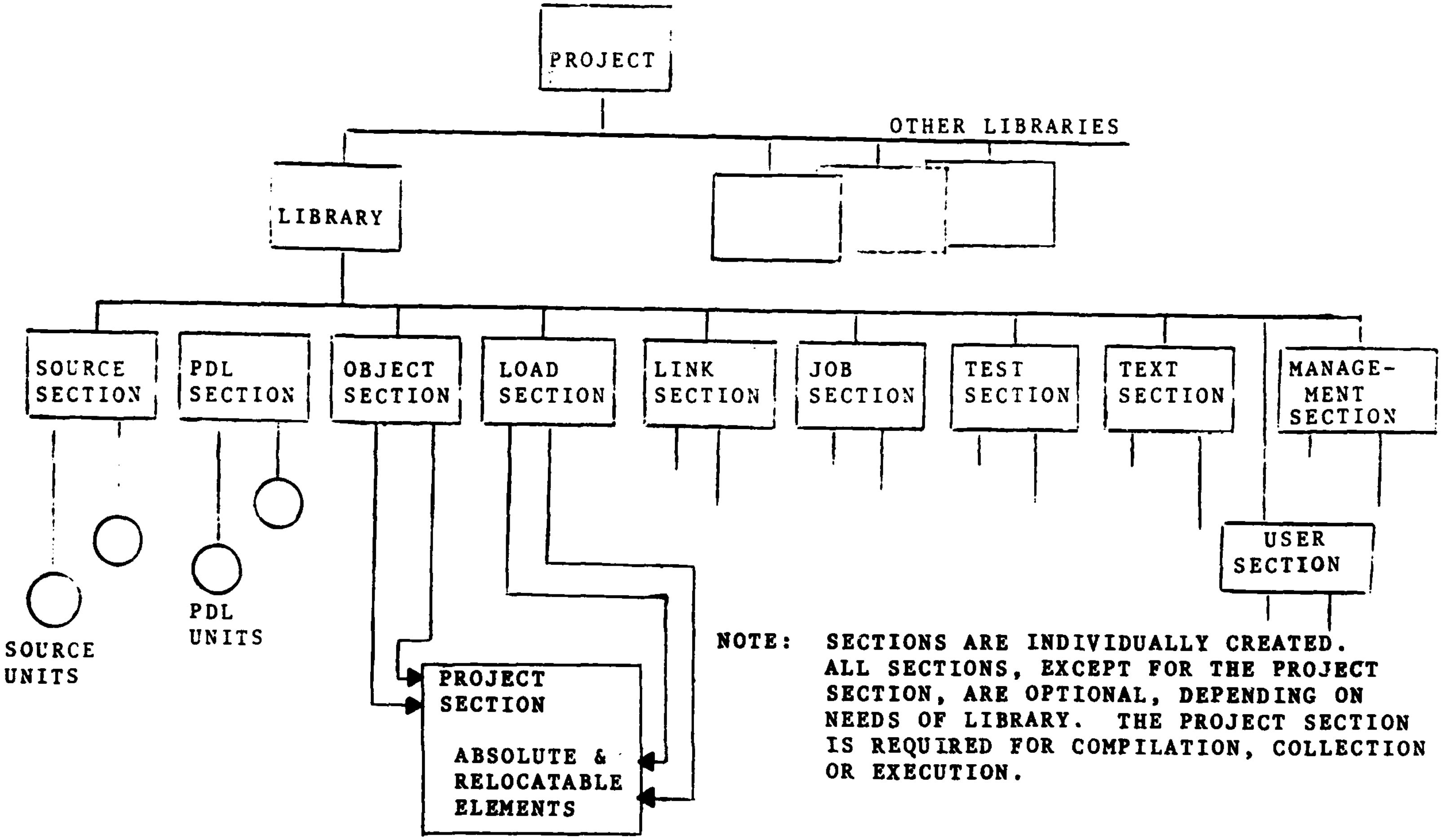


Figure 2-01. PSL System Flowchart



NOTE: SECTIONS ARE INDIVIDUALLY CREATED. ALL SECTIONS, EXCEPT FOR THE PROJECT SECTION, ARE OPTIONAL, DEPENDING ON NEEDS OF LIBRARY. THE PROJECT SECTION IS REQUIRED FOR COMPILATION, COLLECTION OR EXECUTION.

Figure 2-02. Hierarchical Structure of a Project

A library is composed of segments of programming data grouped according to type of data. Each type of data is stored in a specific section of a library. The names of a section is one of the following predefined section names:

a.	SOURCE	-	source code statements
b.	OBJECT	-	object module indexes and accounting records
c.	LOAD	-	load module indexes and accounting records
d.	LINK	-	collector control cards
e.	JOB	-	job control cards used during execution of user program
f.	TEST	-	test data for use by user program
g.	PDL	-	Program Design Language statements
h.	TEXT	-	documentation
i.	MGMT	-	management data
j.	USER	-	user generated data
k.	PROGRAM	-	relocatable and absolute elements

The PSL system manipulates the data for each section appropriately for that type of section.

A section is composed of logical segments of data called units. The unit is the lowest level of the naming convention used under the PSL system. A unit is therefore uniquely identified by the following set:

a. Project name

b. Library name

c. Section name

d. Unit name

The actual structure and content of a unit is related to the type of section to which the unit belongs. See Appendix A for a more detailed discussion of the structure of a section.

To support programmer communication and managerial control, the contents of the project libraries should be available for common reference. The PSL facilitates the parallel maintenance of programming data in both computer-readable and hard-copy form. The system provides the necessary output, in the form of listings and summaries, to ensure that these two libraries may be maintained in exact correspondence. Recommended procedures for updating a library and for filing the output are presented in Section 3.

Section 3. TECHNICAL OPERATION

The PSL system can be used to support a varied set of programmer needs, ranging from a small program to large and complex systems. The user has an extensive selection of options, for which default values are provided where practical. Thus a programmer may begin to use the system in a simple straight-forward manner and gradually enlist the aid of the more specialized services of the system.

Within this section of the Users Manual, the use of the PSL system is addressed at an increasingly detailed technical level. The content of the subsections includes:

a. Overview of Input Requirements (subsection 3.1) - An overall view of the use of the PSL Functions. Details of format are not covered. Each Function is discussed briefly under one of the following programming areas:

(1) Library Maintenance
(2) Unit Maintenance
(3) Program Processing
(4) Output Processing
(5) General Functions
(6) Management Data Collection

b. Composition Guides (subsection 3.2) - Detailed information on miscellaneous general capabilities and requirements which are related to more than one PSL Function. The user should have some understanding of the items discussed in this subsection before attempting to compose an input stream in accordance with the detailed explanations of PSL Functions given in subsection 3.4. The items discussed in section 3.2 are:

(1) High-level parameters
(2) Multi-library search
(3) Independent files
(4) File Management Space
(5) Spawned jobs
(6) Stub generation
(7) Error handling
(8) INCLUDE Statements
(9) Name lengths and Restrictions
(10) Management Data Cycling and Archive Operations
(11) Special Case Card Data

c. Conventions and Glossary (subsection 3.3) - The meanings of formats and the definitions of terms used in Subsection 3.4.

d. Input Format of Function Cards (subsection 3.4) - Detailed explanation of a PSL Function card and of each PSL Function. The Functions are arranged alphabetically.

e. Sample Inputs (subsection 3.5) - References as to locations within this manual where input samples may be found.

f. Procedures for Output (subsection 3.6) - Procedures for maintaining an external library.

3.1 Overview of Input Requirements

The following paragraphs describe input requirements at a fairly elementary level in order to present the basic uses of the PSL system in as simple a manner as possible. Many options and varied approaches are intentionally ignored in this subsection. In addition, abbreviations and short-cuts are not used in the examples, since they would detract from the understanding of the basic Functions. For a fuller and more precise understanding of the use of the system, the user should read the discussion of input formats in subsection 3.2 and the detailed descriptions of the Functions in subsection 3.4.

The input to the PSL system consists of directives on input cards (PSL Function cards), data from the user's libraries, and, in some instances, an input tape. The PSL system accepts PSL Function cards as batch-input data cards. The user's own program and data cards are interspersed, as necessary, with the PSL Function cards. The characters "**ƀ" in columns 1 through 3 identify a card as a PSL Function card or a Function continuation card. The PSL cards contain the name of a Function requested and, if appropriate, sets of keywords and value-entries. Note that the examples given below and in subsection 3.4 represent card input to the PSL system.

In this overview, the PSL Functions are arranged in groups which are related in general capability. The capability groups and the functions under each of these groups are:

a. Library Maintenance

(1) INITIAL - Initialize a project
(2) CREATE - Create a section
(3) BACKUP - Backup
(4) RESTORE - Restore
(5) TERMINATE - Delete a section

b. Unit Maintenance

(1) ADD - Add a unit
(2) REPLACE - Replace a unit
(3) CHANGE - Change a unit
(4) MOVE - Move a unit
(5) PURGE - Delete a unit

c. Program Processing

(1) COMPILE - Compile a module
(2) LINK - Link a program
(3) EXECUTE - Execute a program

d. Output Processing

(1) INDEX - Print a section index
(2) SOURCE - Output a unit
(3) MDPRINT - Print reports
(4) DOCUMENT - Print text
(5) AUTHOR - Print by author
(6) CSCAN - Print by character string

e. General Functions

(1) PARAM - Establish high-level parameters
(2) JCL - Insert extra JCL into spawned job

f. Management Data Collection

(1) MDPLAN - Maintain the data collection and storage plan
(2) MDFORMAT - Maintain management data formats
(3) MDUPDATE - Maintain manual data
(4) MDXCHECK - Maintain exception checking specifications
(5) MDCOLLECT - Collect management data

3.1.1 Library Maintenance

Before programs are developed under the PSL system, a user establishes his project and library. The following PSL Functions are used for general library maintenance.

3.1.1.1 INITIAL

Under the PSL system, a user stores programs and data in discrete units, within sections of a library, under a project. A project must be initialized before libraries are built under it. The PSL Function INITIAL establishes the project as a PSL project and prepares an index for library-sections under the project.

Example:

```
**  INITIAL     PROJECT=FHACD129
```

3.1.1.2 CREATE

After a project is initialized, the CREATE Function is used to build sections in the user's library. The library name uniquely identifies a group of standard sections under a project.

Example:

```
**  CREATE      PROJECT=FHACD129,LIBRARY=NEWCODE,SECTION=SOURCE
**  CREATE      PROJECT=FHACD129,LIBRARY=NEWCODE,SECTION=OBJECT
```

Within a library, the PSL system allows the user to create any of the ten standard PSL sections and a special program section:

a. SOURCE - This section contains all of the source code which is stored in a library, regardless of language or structure. Units of structured code, which at present include Structured COBOL (SCOBOL), JOVIAL (SJOVIAL) and Structured FORTRAN (SPFORT), and units of unstructured source code which are processed by the General Preprocessor (see Appendix I) are stored in blocks within the PSL standard file. It is recommended that the size of structured units be restricted to one page of code. For compilation, both structured units and General Preprocessor units are combined in accordance with the INCLUDE statements found in the units. Units of unstructured code (ASM, COBOL, FORTRAN and JOVIAL) are stored as complete compilable units in sequential files.

b. OBJECT - The object modules which result from compilation of source code are recorded in the OBJECT section. The name of the object unit is the same as the name of the top-most unit of source code. Object modules are used singly or in combination to form a load module for execution.

c. LOAD - If the load module is to be saved on permanent storage, it is recorded as a system-loadable program in the LOAD section. Units in the OBJECT and LOAD sections are produced by Sperry Univac system functions and are therefore not maintained with unit maintenance Functions, as are the other sections.

d. LINK - Units in this section are used to store control cards for directing the loading processing when more than one object module is to be loaded.

e. JOB - Control cards may be stored in units in the JOB section and invoked by the EXECUTE Function. The cards will be added to the program execution activity to provide the job control cards the user's program requires.

f. TEST - This is a general section for card-format data usually used for test input.

g. PDL - Program Design Language statements are stored in units in the PDL section. PDL consists of English-like statements which follow the basic rules of structured programming and are used to define the program structure and logic.

h. TEXT - This section contains units of standard textual material, which are written primarily for use as program documentation. The units may be maintained under the PSL system and printed as any other card-format unit.

i. USER - Units in this section are used to store data generated by non-PSL functions.

j. MGMT - A mangement plan unit is initialized in the created MGMT section and subsequently updated to define the project elements that comprise the management data report requirement. Format units are established to define the report contents and input units are added to introduce manual data. Both automatic data from unit accounting records and manually input data are collected and archived in collection units for subsequent input to management data reports.

k. PROGRAM - Object modules which have been recorded in the OBJECT section are stored as relocatable elements in the PROGRAM section. Load modules which have been recorded in the LOAD section are also stored in the PROGRAM section as absolute modules.

3.1.1.3 BACKUP

The PSL system provides a BACKUP Function to save the user's programs and data on tape. The BACKUP may be invoked for a complete project, a library, or a section.

```
Example:
**  BACKUP     PROJECT=FHACD129,LIBRARY=NEWCODE,SECTION=ALL
(tape identification required)
```

3.1.1.4 RESTORE

The RESTORE Function is used to write the contents of the BACKUP tape into the library. This Function may be used to restore the complete contents of the BACKUP tape, or it may selectively recover a portion of the total data tape at a lower level.

```
Example:
**  RESTORE    PROJECT=FHACD129,LIBRARY=NEWCODE,
               SECTION=SOURCE,UNIT=TIPTOP
(tape identification required)
```

3.1.1.5 TERMINATE

This Function enables a user to delete a section, a library, or an entire project. Files which are released are overwritten.

```
Example:
**  TERMINATE  PROJECT=FHACD129,LIBRARY=NEWCODE,
**             SECTION=OBJECT
```

3.1.2 Unit Maintenance

After a user has initialized a project and has created the sections which are needed in his library, data may be stored in these units. The user may add and update data in those sections which are used for card-image data (SOURCE, PDL, LINK, JOB, TEST, and TEXT). Units in the OBJECT and LOAD sections are not maintained directly by the user, since they are automatically created and updated by the PSL system as a result of the COMPILE and LINK Functions (see subsection 3.1.3). MGMT units are updated by the management data collection Functions. The user may add and update data created by non-PSL functions in the USER section.

The PSL system automatically maintains accounting information for each unit in each section. This information is initialized when the unit is created and updated when the unit is modified. The items of accounting data for each unit are shown in Figure 3-01.

The following PSL functions are used to add, change, and purge units of card-image data in sections (other than MGMT) in the user's library.

3.1.2.1 ADD

The ADD Function is used for the original introduction of data into a unit. The accounting information for the unit is initialized by the ADD Function. The actual data cards for the new unit follow the ADD Function card in the run stream.

```
Example:
**  ADD          PROJECT=FHACD129,LIBRARY=NEWCODE,
**               SECTION=SOURCE,UNIT=TIPTOP,
**               LANGUAGE=SCOBOL
(user's source statements follow)
```

3.1.2.2 REPLACE

After data has been added to a unit, the REPLACE Function is used to completely replace all of the data lines in the unit. Accounting information is not re-initialized, but is updated.

```
Example:
**  REPLACE      PROJECT=FHACD129,LIBRARY=NEWCODE,
**               SECTION=SOURCE,UNIT=TIPTOP
(user's source statements follow)
```

Accounting Information in First Data Block

Accounting Record

Unit type
Including unit name
Version number
Modification level
Date unit was originated
Name of originator
Date of last update
Time of last update
Name of update programmer
Unit key
Unit language
Number of times included
Number of lines in unit

Extended Accounting Record

Number of lines added
Number of lines changed
Number of lines deleted
Total number of lines input to unit
Number of lines input in this cycle
Number of lines copied to unit
Number of times unit was compiled (top)
Total number of updates to unit
Number of updates to unit in this cycle

Figure 3-01. Accounting Information for a Unit

Accounting Information for Management Data Units

Accounting Record

Unit type
Verification status
End of cycle switch
Archive indicator
Start date of cycle
End date of cycle
Cycle period
Cycle count
Archive count
Unit level name
Version name
Modification number
Date unit was originated
Name of originator
Date of last update
Time of last update
Name of update programmer
Unit key
Unit language
Number of times INCLUDED
Number of lines in unit

Figure 3-01. Accounting Information for a Unit (Continued)

3.1.2.3 CHANGE

The CHANGE Function is used to modify the contents of a unit, Modification details are provided on Subfunction cards (COPY, DELETE, INSERT, MODIFY, and SHIFT) which follow the CHANGE card. New source statements follow the INSERT and MODIFY Subfunction cards.

```
Example:
**  CHANGE      PROJECT FHACD129,LIBRARY=NEWCODE,
**              SECTION=SOURCE,UNIT=TIPTOP
**  COPY        AFTER=0,OLDU=ADUN,OLDPROJ=FHACD147,
                FROM=(1,10),
**  SHIFT       LINE=(1,4),COLUMN=R5
**  MODIFY      LINES=(5,9)
(new data inserted here)
**  DELETE      LINES=(11,12)
**  INSERT      AFTER=13
(new data inserted here)
**  MODIFY      LINE=(14,18),FROM=MOVETO,TO="MOVE"
```

3.1.2.4 MOVE

Units are moved from one library to another, or within a library, with the MOVE Function.

```
Example:
**  MOVE        PROJECT=FHACD147,LIBRARY=PROVEN,
**  OLDPROJ=FHACD129,OLDLIB=NEWCODE,
**        SECTION=SOURCE,UNIT=TIPTOP
```

3.1.2.5 PURGE

An individual unit is removed from a library with the PURGE Function. A user should be aware that units are also removed from a library when a higher aggregation, such as a section or library, is removed with the TERMINATE Function.

```
Example:
**  PURGE       PROJECT=FHACD129,LIBRARY=NEWCODE,
**              SECTION=SOURCE,UNIT=TIPTOP
```

If the PURGE of a unit results in the release of a file, the file contents will be overwritten. See subsection 3.4.22 (PURGE).

3.1.3 Program Processing

The PSL system provides many facilities to support the compilation, loading, and execution of programs. One of the more powerful capabilities available for these Functions is the optional search through multiple libraries during retrieval. This option is illustrated under the appropriate Functions in subsection 3.4. In this subsection each of the three program processing Functions is presented in its most straightforward form.

3.1.3.1 COMPILE*

The COMPILE Function retrieves units from the SOURCE section, invokes the appropriate precompiler for structured source code, compiles (or assembles) the resulting stream of code, records the unit in the OBJECT section, and stores the compiled source as a relocatable element in the PROGRAM section. The UNIT name is the name of the top-most source unit of the module which is to be compiled. This name will be used for the name of the compiled OBJECT unit. The processing which is invoked by the COMPILE Function depends upon the language of the unit. Under the present PSL system, procedures exist to process languages ASM, ASMG (compacted ASM), COBOL, COBOLG (compacted COBOL), SCOBOL (Structured COBOL), FORTRAN, FORTRANG (compacted FORTRAN), SPFORT (Structured FORTRAN), JOVIAL, JOVIALG (compacted JOVIAL), and SJOVIAL (Structured JOVIAL).

Example:

```
** COMPILE     PROJECT=FHACD147,LIBRARY=NEWCODE,UNIT=TIPTOP
```

3.1.3.2 LINK

The LINK Function retrieves one or more object module entries from the OBJECT section, collects the corresponding relocatable elements, and records the new absolute element in the LOAD section. The resulting absolute element is stored as a unit in the PROGRAM section.

Example:

```
** LINK        PROJECT=FHACD147,LIBRARY=NEWCODE,LINE=TIPTOP
```

If more than one object module is to be linked together to form the load module, collector control cards are retrieved from a unit in the LINK section. This option, and others, are explained under the LINK Function in subsection 3.4.13.

*Also see Appendix I.

3.1.3.3 EXECUTE

The user's job control cards for execution of his programs are previously stored as a unit in the JOB section. The EXECUTE Function invokes the execution of his program using these job control cards as his input run stream.

The program itself may be retrieved by the PSL system in one of the following forms:

a. A load module from the PROGRAM section

b. A series of one or more object modules which are selected as a result of collector control cards in a unit of the LINK section and which will be processed by the Collector.

Example:

```
**  EXECUTE          PROJECT=FHACD147,LIBRARY=NEWCODE,
**                   JOB=MYJCL,LINK=TIPTOP
```

See subsection 3.4 for a more detailed explanation.

3 .1.4 Output Processing

Six PSL Functions are available in the PSL system to obtain reports.

3.1.4.1 INDEX

The INDEX Function prints a status report for a section. The report contains information pertaining to the section as a whole, as well as a listing of the index for the section. An example of a Section Index Report is shown in Figure 3-02.

Example:

```
**  INDEX      PROJECT=FHACD147,LIBRARY=NEWCODE,SECTION=SOURCE
```

3.1.4.2 SOURCE

The SOURCE Function can be used to print a listing of any unit which is in card-image format. This includes the units in the SOURCE, PDL, LINK, JOB, MGMT, TEST, and TEXT sections. If the unit is from the SOURCE or the PDL section and the unit is written in a supported structured language (Structured COBOL, JOVIAL and FORTRAN), the structured code is automatically indented for the proper alignment of the structures on the listing. This listing is always provided when a unit is added to the library or modified. The contents of the unit may, on option, be written to tape or punched on cards. In these latter cases, the header information is omitted and the code is

```
                                        INCLUDED                                UPDATED
UNIT                                TYPE       COUNT VER/MOD UPDATED               BY      LANGUAGE LINES CREATED
*************************** ************* ******* ************** ******* ******** ***** ********

PROCESS-FUNCTION                    STUB           1 000/000 10/07/76 12109 PSLTEST SCOBOL       0 10/07/76
SPAWN-A-JOB                         STUB           1 000/000 10/07/76 12109 PSLTEST SCOBOL       0 10/07/76
TIPTOP                              MAIN             001/000 10/07/76 12109 PSLTEST SCOBOL      14 10/07/76
TIPTOP-ENVIRONMENT-DIVISION INCLUDED       1 001/000 10/07/76 12109 PSLTEST SCOBOL      25 10/07/76
TIPTOP-FILE-SECTION                 INCLUDED       1 001/000 10/07/76 12109 PSLTEST SCOBOL      10 10/07/76
TIPTOP-PROCEDURE-DIVISION           INCLUDED       1 001/000 10/07/76 12109 PSLTEST SCOBOL      41 10/07/76
TIPTOP-WORKING-STORAGE              INCLUDED       1 001/000 10/07/76 12109 PSLTEST SCOBOL      42 10/07/76
```

Figure 3-02. Section Index Report

reproduced as it exists in the unit, without automatic indentation. An example of a listing of structured COBOL (SCOBOL) is shown in Figure 3-03.

```
Example:
**  SOURCE       PROJECT=FHACD129,LIBRARY=NEWCODE,
**               SECTION=SOURCE,UNIT=TIPTOP-PROCEDURE-DIVISION
```

3.1.4.3 MDPRINT

The MDPRINT Function is used to print management data reports. There are two categories of such reports:

a. Program Structure Report
b. Management Data Report

The Program Structure (PS) report begins with the unit which is named on the Function card and develops a hierarchically nested and indented list of all units which are referenced by or in units which are themselves INCLUDEd in the hierarchical program structure. Units which are referenced by a CALL statement are also printed with the appropriate indentation in the list, but they are not automatically searched for lower levels of INCLUDE or CALL statements. An option is available to invoke a PS report for all CALLed units. Statistical organization information is printed for each unit. An alphabetically-arranged list of all referenced units follows the hierarchical list. An example of a Program Structure Report is shown in Figure 3-04.

```
Example:
**  MDPRINT      PROJECT=FHACD129,LIBRARY=NEWCODE,
**               UNIT=TIPTOP,REPORT=PS
```

A Management Data report prints the contents of one or more management data units. The top-level unit is identified by the UNIT keyword. Management data units may contain a combination of automatically collected data and manually input data combined according to the specifications in an associated user-defined format unit. A general format is provided to accommodate a wide variety of data collections to be printed. A representative Management Data Report is shown in Figure 3-05.

```
Example:
**  MDPRINT      PROJECT=FHACD129,LIBRARY=NEWCODE,
**               UNIT=TIPTOP,REPORT=MD
```

```
  BATCH-PSL-CONTROL.
1  DISPLAY " TRACE - BCTL (BATCH-PSL-CONTROL) EXECUTED.".
   MOVE SPACES TO PARAMETER-TABLE.
   CALL OBUSE
         USING USERID-FROM-CARD. IDENT-INFO.
   MOVE USERID-FROM-CARD TO
         USERID OF PARAMETER-TABLE.
         PROGRAMMER-NAME OF PARAMETER-TABLE.
   OPEN INPUT INPUT-CARDS
         OUTPUT MESSAGE-FILE.
   MOVE "ADD" TO SHORT-INPUT-FUNCTION
   READ INPUT-CARDS
     AT END
       MOVE ZERO TO PROCESSING-STATUS.
   IF PROCESSING-STATUS NOT EQUAL TO CODE-FOR-END-INPUT-CARDS
     CALL OBFN
           USING INPUT-CARD-FUNCTION. PROCESSING-STATUS.
     DO UNTIL NORMAL-STATUS
       MOVE HIGH-FUNCTION-VALUE TO FUNCTION-NUMBER.
       SEARCH ALL PSL-FUNCTIONS
         WHEN PSL-FUNCTION (PF-INDEX) EQUAL TO
                 SHORT-INPUT-FUNCTION
           SET FUNCTION-NUMBER TO PF-INDEX.
       INCLUDE PROCESS-FUNCTION.
       CALL OBFN
             USING INPUT-CARD-FUNCTION. PROCESSING-STATUS.
     ENDDO
     MOVE CODE-FOR-END-INPUT-CARDS TO MESSAGE-NUMBER.
     CALL PRMS
           USING MESSAGE-NUMBER. MESSAGE-DATA.
     IF SPAWN-JOB
       INCLUDE SPAWN-A-JOB.
     ENDIF
   ELSE
     MOVE CODE-FOR-NO-INPUT-CARDS TO MESSAGE-NUMBER.
     CALL PRMS
           USING MESSAGE-NUMBER. MESSAGE-DATA.
   ENDIF
   CALL RLAF
   CLOSE INPUT-CARDS. MESSAGE-FILE.
   STOP RUN.
```

Figure 3-03. Unit Listing with Automatic Indentation

```
UNIT  UNIT                                                         ORIGINATE   LAST     PROJECT  LIBRARY  SP
LEVEL LINES      UNIT NAME                      UNIT TYPE            DATE     UPDATE     NAME     NAME   FLAG
----- ----- ------------------------------ --------------- --------- --------- -------- ------- ----

  1     14  TIPTOP                         MAIN            07 OCT 76 07 OCT 76 FHACD129 NEWCODE
  2     23    TIPTOP-ENVIRONMENT-DIVISION  REAL-SINGLE-INCL 07 OCT 76 07 OCT 76     *        *
  2     16    TIPTOP-FILE-SECTION          REAL-SINGLE-INCL 07 OCT 76 07 OCT 76     *        *
  2     42    TIPTOP-WORKING-STORAGE       REAL-SINGLE-INCL 07 OCT 76 07 OCT 76     *        *
  2     41    TIPTOP-PROCEDURE-DIVISION    REAL-SINGLE-INCL 07 OCT 76 07 OCT 76     *        *
  3             OBUSE                      CALLED
  3             OBFN                       CALLED
  3             PROCESS-FUNCTION           STUB-SINGLE-INCL 07 OCT 76 07 OCT 76 FHACD129 NEWCODE
  3             OBFN                       CALLED
  3             PRMS                       CALLED
  3             SPAWN-A-JOB                STUB-SINGLE-INCL 07 OCT 76 07 OCT 76 FHACD129 NEWCODE
  3             PRMS                       CALLED
  3             RLAF                       CALLED



SP-FLAG-MESSAGES (NBRS, UNDER SP-FLAG HEADER ARE CUMULATIVE)          * - SAME NAME
    1 - UNIT CONTAINS MOD TO- STATEMENTS
    2 - NBR. OF LINES EXCEEDS DEFINED LIMITS
    4 - MORE THAN ONE STATEMENT ON A LINE
```

Figure 3-04. Program Structure Report
(Part 1 of 2)

```
                              CROSS REFERENCE LISTING

        UNIT NAME            PROJECT NAME LIBRARY NAME  UNIT TYPE           HIGHER UNIT NAME
---------------------------- ------------ ------------ ----------------  -------------------------

OBFN                                                   CALLED            TIPTOP-PROCEDURE-DIVISION
OBUSE                                                  CALLED            TIPTOP-PROCEDURE-DIVISION
PRMS                                                   CALLED            TIPTOP-PROCEDURE-DIVISION
PROCESS-FUNCTION               FHACD129     NEWCODE    STUB-SINGLE-INCL  TIPTOP-PROCEDURE-DIVISION
RLAF                                                   CALLED            TIPTOP-PROCEDURE-DIVISION
SPAWN-A-JOB                    FHACD129     NEWCODE    STUB-SINGLE-INCL  TIPTOP-PROCEDURE-DIVISION
TIPTOP                            *            *       MAIN              TOP OF TREE
TIPTOP-ENVIRONMENT-DIVISION       *            *       REAL-SINGLE-INCL  TIPTOP
TIPTOP-FILE-SECTION               *            *       REAL-SINGLE-INCL  TIPTOP
TIPTOP-PROCEDURE-DIVISION         *            *       REAL-SINGLE-INCL  TIPTOP
TIPTOP-WORKING-STORAGE            *            *       REAL-SINGLE-INCL  TIPTOP
```

Figure 3-04. Program Structure Report
(Part 2 of 2)

```
                                                                                         ITEM
-----------------ITEM LABEL-------------------  ----------ITEM VALUE-------------- ITEM NAME    NBR
+++++++++++++++++++++++++++++++++  SYSTEM  ---- NEWCODE                  ++++++++++++++

PROJECT TITLE                                   NEWCODE-TITLE                      PROJ-TITLE    010
PROJECT DESCRIPTION                             NEWCODE PROJECT DESCRIPTION LINE 1 PROJ-DESCR    020
PROJECT START DATE                                770501                           START-DATE    030
ESTIMATED COMPLETION DATE                         770901                           EST-END-DATE  040
ACTUAL COMPLETION DATE                                 0                           ACT-END-DATE  050
PLANNED AVERAGE YEARS EXPERIENCE MANAGERS             10                           P-AVE-MGRS    060
PLANNED AVERAGE YEARS EXPERIENCE ANALYSTS             12                           P-AVE-ANAL    070
PLANNED AVERAGE YEARS EXPERIENCE PROGRAMMER            5                           P-AVE-PROG    080
PLANNED AVERAGE YEARS EXPERIENCE ADMINISTRATIVE        2                           P-AVE-ADMIN   090
PLANNED AVERAGE YEARS EXPERIENCE OTHER                 0                           P-AVE-OTH     100
ACTUAL AVERAGE YEARS EXPERIENCE MANAGERS               8                           A-AVE-MGRS    110
ACTUAL AVERAGE YEARS EXPERIENCE ANALYSTS              11                           A-AVE-ANAL    120
ACTUAL AVERAGE YEARS EXPERIENCE PROGRAMMERS            7                           A-AVE-PROG    130
ACTUAL AVERAGE YEARS EXPERIENCE ADMINISTRATIVE         3                           A-AVE-ADMIN   140
ACTUAL AVERAGE YEARS EXPERIENCE OTHER                  1                           A-AVE-OTH     150
PLANNED NUMBER OF MANAGERS                             4                           P-NBR-MGRS    160
PLANNED NUMBER OF ANALYSTS                             5                           P-NBR-ANAL    170
PLANNED NUMBER OF PROGRAMMERS                         16                           P-NBR-PROG    180
PLANNED NUMBER OF ADMINISTRATIVE                       4                           P-NBR-ADMIN   190
PLANNED NUMBER OF OTHER                                2                           P-NBR-OTH     200
ACTUAL NUMBER OF MANAGERS                              4                           A-NBR-MGRS    210
ACTUAL NUMBER OF ANALYSTS                              4                           A-NBR-ANAL    220
ACTUAL NUMBER OF PROGRAMMERS                          12                           A-NBR-PROG    230
ACTUAL NUMBER OF ADMINISTRATIVE                        2                           A-NBR-ADMIN   240
ACTUAL NUMBER OF OTHER                                 1                           A-NBR-OTH     250
ESTIMATED PERSONNEL TURNOVER RATE                      4                           E-TURNOVER    260
ACTUAL PERSONNEL TURNOVER RATE                         0                           A-TURNOVER    270
ESTIMATED LOCAL TRAVEL CHANGE                         30                           E-LOC-TRIPS   280
ACTUAL LOCAL TRAVEL                                   60                           A-LOC-TRIPS   290
ESTIMATED DISTANT TRAVEL                               6                           E-DIS-TRIPS   300
ACTUAL DISTANT TRAVEL                                  0                           A-DIS-TRIPS   310
ESTIMATED WORKING CONDITIONS                           5                           E-WORK-COND   320
ACTUAL WORKING CONDITIONS                              6                           A-WORK-COND   330
PLANNED PROGRAMMING LANGUAGE EXPERIENCE                3                           P-LANG-EXP    340
ACTUAL PROGRAMMING LANGUAGE EXPERIENCE                 5                           A-LANG-EXP    350
PLANNED SIMILAR APPLICATION EXPERIENCE                 3                           P-SIM-EXP     360
ACTUAL SIMILAR APPLICATION EXPERIENCE                  6                           A-SIM-EXP     370
PLANNED TARGET COMPUTER EXPERIENCE                     3                           P-TARG-EXP    380
ACTUAL TARGET COMPUTER EXPERIENCE                      5                           A-TARG-EXP    390
ESTIMATED CUSTOMER APPLICATION EXPERIENCE              3                           E-APPL-EXP    400
ACTUAL CUSTOMER APPLICATION EXPERIENCE                 4                           A-APPL-EXP    410
ESTIMATED CUSTOMER EQUIPMENT EXPERIENCE                2                           E-EQUIP-EXP   420
ACTUAL CUSTOMER EQUIPMENT EXPERIENCE                   1                           A-EQUIP-EXP   430
```

Figure 3-05. Management Data Report

3.1.4.4 AUTHOR

The AUTHOR Function can be used to print a list of unit names, or listings of the actual units, which were originally generated by and/or updated by a specific programmer. An example of an AUTHOR Report is shown in Figure 3-06.

```
Example:
**  AUTHOR      PROJECT=FHACD129,LIBRARY=NEWCODE,
**              SECTION=SOURCE,UPGMR=SMITH,OPTION=SOURCE
```

3.1.4.5 DOCUMENT

The DOCUMENT Function is used to print documentation stored in a library in the form of program design language, structured source code, text, etcetera. Output requirements are specified through keyword options provided on subfunction cards (HEADER, TEXT, EJECT, and SPACE). An example of a DOCUMENT Report is shown in Figure 3-07.

```
Example:
**  DOCUMENT    PROJ=FHACD129,LIB=NEWCODE,
**              SECTION=SOURCE
**  HEADER
    THIS IS A HEADER CARD
**  TEXT        UNIT=FIRST-UNIT
**  SPACE       LINES=6
**  TEXT
    THIS IS CARD 1 OF 2
    THIS IS CARD 2 of 2
**  EJECT
**  TEXT        UNIT=LAST-UNIT
```

3.1.4.6 CSCAN

The CSCAN Function is used to scan all units of the indicated section to locate a specific character string. The string may be up to 48 characters in length. The resulting output will be a list of the unit names containing the specific character string and the corresponding lines of code for each occurrence. Figure 3-08 contains an example of character scan output.

```
Example:
**  CSCAN       PROJ=FHACD129,LIBR=NEWCODE,
**              SECTION=JOB,STRING=TIPTOP
```

```
                                       INCLUDED
UNII                           TYPE       COUNT VER/MOD UPDATED        PROGRAMMER   LANGUAGE LINES CREATED
****************************** **************** ******* ************** ************ ******** ***** ********

FMED-WORKING-STORAGE-77        INCLUDED       1 015/000 05/11/77 14:16 UPD/DGC      SCOBOL      51 05/10/77
FMMDE                          MAIN             030/000 05/10/77 20:06 ORG/FHACD128 SCOBOL      18 04/20/77
FMMD-ACCESS-MGMT-DATA-FILE     INCLUDED       1 033/000 05/11/77 14:16 UPD/DGC      SCOBOL      37 05/11/77
FMMD-CALL-MOVE-OLDPROJ-OLDLIB  INCLUDED       1 003/000 05/11/77 14:16 UPD/DGC      SCOBOL       6 05/11/77
FMMD-CHECK-FMMD-VALUES         INCLUDED       1 035/000 05/11/77 14:16 UPD/DGC      SCOBOL      41 05/11/77
FMMD-CLEANUP-WRITE-ACCT-INFO   INCLUDED       1 003/000 05/11/77 14:16 UPD/DGC      SCOBOL      52 05/11/77
FMMD-DELETE-MGMT-INFO          INCLUDED       1 003/000 05/11/77 14:17 UPD/DGC      SCOBOL      16 05/11/77
FMMD-ENVIRONMENT-DIVISION      INCLUDED       1 034/000 05/11/77 14:17 UPD/DGC      SCOBOL      20 05/10/77
FMMD-FILE-SECTION              INCLUDED       1 034/000 05/11/77 14:17 UPD/DGC      SCOBOL      15 05/10/77
FMMD-INTERPRET-DATA-VALUE      INCLUDED       1 034/000 05/11/77 14:17 UPD/DGC      SCOBOL      72 05/11/77
FMMD-INTERPRET-FMMD-VALUE      INCLUDED       1 032/000 05/11/77 14:17 UPD/DGC      SCOBOL      25 05/11/77
FMMD-NUMERIC-EDIT              INCLUDED       2 018/000 05/11/77 14:17 UPD/DGC      SCOBOL      14 05/11/77
FMMD-PROCEDURE-DIVISION        INCLUDED       1 041/000 05/11/77 14:17 UPD/DGC      SCOBOL      62 05/10/77
FMMD-READ-EDIT-SORT-TRANS      INCLUDED       1 014/000 05/11/77 14:17 UPD/DGC      SCOBOL      32 05/11/77
FMMD-SETUP-ACCOUNTING-INFO     INCLUDED       1 005/000 05/11/77 14:18 UPD/DGC      SCOBOL      15 05/11/77
FMMD-STORE-EDIT-KEYWORDS       INCLUDED       1 038/000 05/11/77 14:18 UPD/DGC      SCOBOL      41 05/11/77
FMMD-WORKING-STORAGE-01        INCLUDED       1 054/000 05/11/77 14:18 UPD/DGC      SCOBOL     110 05/10/77
FMMD-WORKING-STORAGE-77        INCLUDED       1 043/000 05/11/77 14:19 UPD/DGC      SCOBOL      26 05/10/77
FMMVE                          MAIN             003/000 05/11/77 14:19 ORG/FHACD128 SCOBOL      18 04/20/77
FMMV-ENVIRONMENT-DIVISION      INCLUDED       1 004/000 05/11/77 14:19 UPD/DGC      SCOBOL      18 05/11/77
FMMV-FILE-SECTION              INCLUDED       1 005/000 05/11/77 14:19 UPD/DGC      SCOBOL      15 05/11/77
FMMV-MOVE-OLDPROJ-OLDLIB       INCLUDED       1 005/000 05/11/77 14:19 UPD/DGC      SCOBOL      50 05/11/77
FMMV-PROCEDURE-DIVISION        INCLUDED       1 005/000 05/11/77 14:19 UPD/DGC      SCOBOL      15 05/11/77
FMMV-WORKING-STORAGE-01        INCLUDED       1 006/000 05/11/77 14:19 UPD/DGC      SCOBOL     177 05/11/77
FMMV-WORKING-STORAGE-77        INCLUDED       1 003/000 05/11/77 14:19 UPD/DGC      SCOBOL      51 05/11/77
FMUPE                          MAIN             023/000 05/10/77 20:07 ORG/FHACD128 SCOBOL      18 04/20/77
FMUP-ADD-CHANGE-TEMP-DATA      INCLUDED       1 012/000 05/11/77 14:20 UPD/DGC      SCOBOL      34 05/11/77
FMUP-CHANGE-MGMT-UNIT          INCLUDED       1 027/000 05/11/77 14:20 UPD/DGC      SCOBOL      30 05/11/77
FMUP-ENVIRONMENT-DIVISION      INCLUDED       1 025/000 05/11/77 14:20 UPD/DGC      SCOBOL      18 05/10/77
FMUP-FILE-SECTION              INCLUDED       1 024/000 05/11/77 14:20 UPD/DGC      SCOBOL      15 05/10/77
FMUP-INITIALIZE-ADD-MGMT-UNIT  INCLUDED       1 013/000 05/11/77 14:20 UPD/DGC      SCOBOL      15 05/11/77
FMUP-PROCEDURE-DIVISION        INCLUDED       1 024/000 05/11/77 14:20 UPD/DGC      SCOBOL      27 05/10/77
FMUP-WORKING-STORAGE-01        INCLUDED       1 033/000 05/11/77 14:20 UPD/DGC      SCOBOL     177 05/10/77
FMUP-WORKING-STORAGE-77        INCLUDED       1 031/000 05/11/77 14:20 UPD/DGC      SCOBOL      51 05/10/77
ITCLE                          CALLED           001/000 05/16/77 19:49 ORG/FHACD128 SCOBOL      39 04/20/77
ITCL-CHECK-COLLECTION-UNIT-KEY INCLUDED       1 001/000 05/16/77 19:49 UPD/DGC      SCOBOL      16 05/16/77
ITCL-INTERPRET-KEYWORD-VALUES  INCLUDED       1 001/000 05/16/77 19:49 UPD/DGC      SCOBOL      53 05/16/77
ITCL-PROCEDURE-DIVISION        INCLUDED       1 001/000 05/16/77 19:50 UPD/DGC      SCOBOL      39 05/16/77
ITCL-STORE-EDIT-INPUT-KEYWORDS INCLUDED       1 001/000 05/16/77 19:50 UPD/DGC      SCOBOL      41 05/16/77
ITCL-WORKING-STORAGE-01        INCLUDED       1 001/000 05/16/77 19:50 UPD/DGC      SCOBOL      65 05/16/77
ITCL-WORKING-STORAGE-77        INCLUDED       1 001/000 05/16/77 19:50 UPD/DGC      SCOBOL      21 05/16/77
ITCL-WRITE-INPUT-KEYWORD-CARDS INCLUDED       1 001/000 05/16/77 19:50 UPD/DGC      SCOBOL      26 05/16/77
ITCL-WRITE-JCL-FOR-MDCOLLECT   INCLUDED       1 001/000 05/16/77 19:51 UPD/DGC      SCOBOL      49 05/16/77
```

Figure 3-06. AUTHOR Report

```
*    THE MODULE TIPTOP PROCESSES THE ADD A UNIT FUNCTION (ADUN)
 TIPTOP
     INITIALIZE PARAMETER-TABLE WITH SPACES
     CALL OBUS
     MOVE PGMR-NAME TO PARAMETER-TABLE
     OPEN INPUT INPUT-CARDS.
           OUTPUT MESSAGE-FILE
     MOVE "ADD" TO INPUT-FUNCTION
     READ INPUT-CARDS
     IF NOT END OF INPUT CARDS
       CALL OBFN.
       DO UNTIL NORMAL RETURN FROM OBFN
         SEARCH PSL-FUNCTIONS
           WHEN PSL-FUNCTION IN THE TABLE EQUAL INPUT-FUNCTION
           SET FUNCTION-NUMBER TO INDEX OF TABLE.
         INCLUDE PROCESS-FUNCTION.
         CALL OBFN.
       ENDDO.
       CALL PRINT MESSAGE (END OF INPUT CARDS).
       IF SPAWN JOB NEEDED
         INCLUDE SPAWN-A-JOB
       ENDIF.
     ELSE
       CALL PRINT MESSAGE (NO INPUT CARDS)
     ENDIF.
     CALL RLAF.
     CLOSE INPUT-CARDS.
           MESSAGE-FILE.
     STOP RUN.
```

Figure 3-07. DOCUMENT Report

```
UNIT                         LINE COL
***************************  **** ***

TIPTOP                          2 23    PROGRAM-ID.  TIPTOP.
                                4 20      INCLUDE TIPTOP-ENVIRONMENT-DIVISION.
                                8 20      INCLUDE TIPTOP-FILE-SECTION.
                               10 20      INCLUDE TIPTOP-WORKING-STORAGE.
                               14 20      INCLUDE TIPTOP-PROCEDURE-DIVISION.
TIPTOP-ENVIRONMENT-DIVISION  NONE
TIPTOP-FILE-SECTION          NONE
TIPTOP-PROCEDURE-DIVISION    STUB
TIPTOP-WORKING-STORAGE       STUB
```

Figure 3-08. Character Scan Output

3.1.5 General Functions

Two PSL Functions are available which have general application in conjunction with other Functions.

3.1.5.1 PARAM

This Function is used to enter high-level parameters which apply to a series of subsequent Functions. The particular parameters which may appear on a PARAM card (PROJECT, LIBRARY, SECTION, PASSWORD, and PGMR) are cumulative parameters. When any of these parameters are established, it remains in effect. A PARAM Function card is ordinarily used to establish these values, but the keywords may be established by Function cards for which they are valid parameters.

```
Example:
**  PARAM      PROJECT=FHACD147,LIBRARY=NEWCODE
**  CREATE     SECTION=SOURCE
**  ADD        UNIT=TIPTOP,LANGUAGE=SCOBOL
(user's source data cards)
```

3.1.5.2 JCL

The JCL Function enables the user to introduce JCL cards into the input stream of a subsequently spawned job. The Function is not required in the basic use of the PSL system, but provides the flexibility of using additional Exec 8 control cards in conjunction with such PSL Functions as COMPILE and LINK. See subsection 3.2 for a discussion of spawned jobs and subsection 3.4.12 for specific examples of the use of this Function.

3.1.6 Management Data Collection

This section contains an overview of the management data collection capability; this overview is intended to simply acquaint the user with the capability. The types of units which are contained in a Management (MGMT) section are identified. The possible levels for which management data can be collected and reported are introduced. The actions necessary to perform a data collection are briefly described.

General Overview

The Management Data Collection and Reporting (MDCR) capability enables software projects utilizing the PSL facility to obtain statistical data reports on the status and progress of project activities. Figure 3-09(a) illustrates a typical MDCR Data Base composed of a single Management (MGMT) section library and multiple Source section libraries. The set of top units and included units presented in these libraries constitutes the total code under development for a given project. Statistics relating to unit update activity are maintained in the individual Unit Accounting records (refer to Figure 3-01 for a list of statistics maintained). These "automatic" statistics may be collected and summarized as directed by the PSL user through Managemenc Data Functions made available with the MDCR capability.

The PSL MGMT section shown in Figure 3-09(a) provides storage for the management data units that direct the MDCR activities. The MDPLAN Function is first utilized to describe the hierarchical organization of data reports whose generation is illustrated in Figure 3-09(b). The MDPLAN input is made in the form of keyword and keyword value specifications which are ordered to reflect the subordinate relationships that are observed in producing data summaries. The hierarchy of data reports shown in Figure 3-09(b) as a result of the given plan input.

The specific items of data to be reported at each level of output is determined by input provided to the MDFORMAT Function. An MDCR Format unit must be established and updated for each of the reported data levels described in the management data plan to reflect the order and source of data items to be collected and summarized. A unit level format may optionally be pro/ided to direct that unsummarized unit accounting record data to be collected and made available for output in conjunction with the "including" module level elements specified in the management data plan.

Module level data may be derived from essentially two data sources; that is, 1) unit accounting records, and 2) manual inputs. The latter source of data input is established through the MDUPDATE Function by adding a management input unit with the same name as the plan element (e.g., module level element) with which the data is to be corresponded. The module level format unit would in turn prescribe that specific data items in the module report are

IBM-G-5

<table>
<tr><td>Applicable
PSL Functions</td><td colspan="6">* * * * * PSL MGMT Section * ∴ * * *</td></tr>
<tr><td rowspan="2">MDPLAN)</td><td colspan="6">Management Data Plan Unit (MDCR-PLAN)</td></tr>
<tr><td colspan="6">SYSTEM = BIGTOP
JOB = BIGJOB
SUBSYS=SUB1, MODULE=MOD1A, MODULE=MOD1B
SUBSYS=SUB2, MODULE=MOD2A, MODULE=MOD2B</td></tr>
<tr><td rowspan="2">MDFORMAT</td><td colspan="6">Management Data Format Units (MDCR-FORMAT-)</td></tr>
<tr><td>SYSTEM</td><td>SUBSYS</td><td>MODULE</td><td>JOB</td><td colspan="2">UNIT</td></tr>
<tr><td rowspan="2">MDUPDATE
MDXCHECK</td><td colspan="6">Management Data Input Units</td></tr>
<tr><td>BIGTOP</td><td>BIGJOB</td><td>SUB1</td><td>SUB2</td><td>MODIA</td><td>ETC.</td></tr>
<tr><td rowspan="2">MDCOLLECT
PURGE</td><td colspan="6">Management Data Collection Units</td></tr>
<tr><td colspan="3">MDCR-COLLECTION</td><td colspan="3">MDCR-ARCHIVE-001-(DATE1)
MDCR-ARCHIVE-002-(DATE2)
MDCR-ARCHIVE-003-(DATE3)</td></tr>
<tr><td colspan="7">* * * * * * PSL SOURCE Section(s) * * * * * *</td></tr>
<tr><td></td><td colspan="3">Top Units</td><td colspan="3">Included Units</td></tr>
<tr><td>ADD
CHANGE
MOVE
REPLACE
PURGE</td><td colspan="3">MOD 1A
MOD 1B
MOD 2A
MOD 2B</td><td colspan="3"></td></tr>
</table>

Figure 3-09(a). Management Data Base Structure

Management Plan Input vs. Management Reports Output

```
System = PSL-project
   Job = PSL-system-test
 Subsystem = PSL-main
   Module = BCTLE
   Module = PPLKE

              .
              .
              .
              .
              .
              .

 Subsystem = PSL-Function
   Module = ADUNE
   Module = CHUNE
   Module = CRFLE

              .
              .
              .
              .
              .
              .

 Subsystem = PSL-auxiliary
   Module = ADXEE
   Module = CHXXE

              .
              .
              .
```

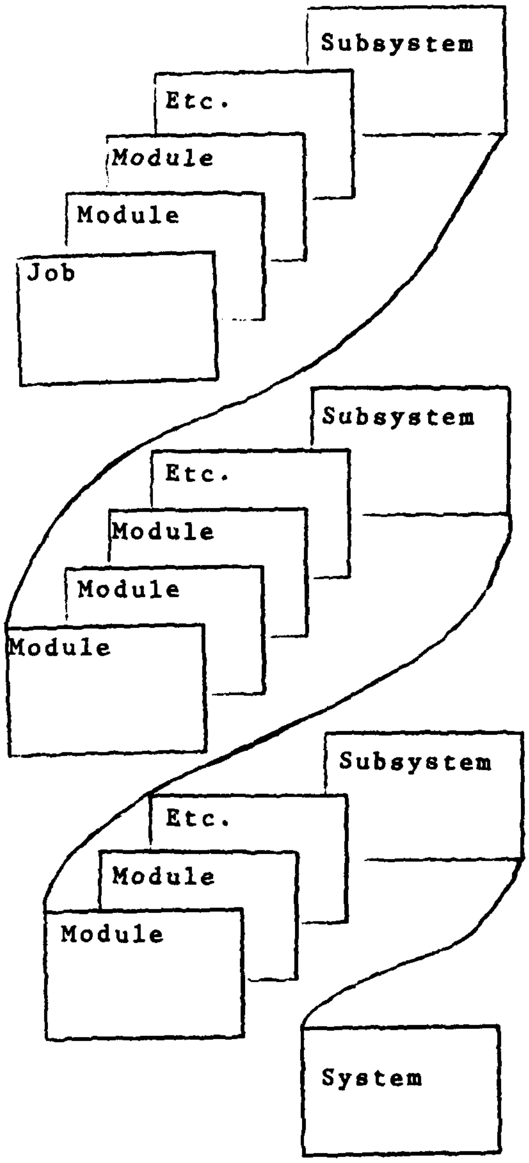


Figure 3-09(b). Management Report Structure to be derived from a manual data source. These items are looked for (whenever module data is collected) in a management input unit whose name corresponds with the module being processed. Provision of such data is optional; "null" values will be reported in the absence of available data.

Job level elements (whose input is currently derived from manual data sources only) may be used to introduce computer utilization statistics (or other pertinent data) into the reported output. Subsystem level data may be derived from subordinate module and/or job level data items and a corresponding management input unit. The subsystem level format reflects the data source of each item listed and may specify the type of summarization to be performed. (Refer to paragraph 3.4.15 for further details). The system level format unit may specify that data be derived from subordinate subsystem, module and/or job level data items as well as corresponding manual data sources.

Management input units established via the MDUPDATE Function may also be used to maintain exception check specifications through use of the MDXCHECK Function. Any two numeric data items specified in the corresponding level format unit may be subjected to a value "variance" check. The data item values are compared at the time the data is collected and a determination made as to whether an exception exists. If an exception does exist, pertinent information is added to the data collection unit so that management reports which are subsequently produced from that collection may flag the excepted data item.

The MDCOLLECT Function performs a data collection when directed to a project MGMT section containing an appropriate management data plan. The ensuing collection activity automatically verifies and cross-relates plan level elements, format requirements and data source availability. If no significant error (e.g., absence of a required format or a plan syntax error) is determined, the data sources indicated in the management data plan and format units will be searched. Source code modules (i.e., top units) which are not found will be noted as errors, but data collection will proceed until the entire management plan is processed and the collected data is stored in the designated MGMT section.

If a previous data collection exists, it will be replaced or archived based upon an "automatic" archive determination or user-specification of a "manual" archive option in conjunction with the MDCOLLECT Function. (Refer to paragraph

3.2.10 for a more detailed discussion of automatic and manual archive determinations). Each archived data collection is named with a unique serial number and date-of-collection suffix as indicated in Figure 3-09(a).

Special Features

More than one report hierarchy may be specified in the management data plan. Each report hierarchy (starting with a unique system level element) will be processed in the sequence that it occurs in the management plan. Alternative aggregations of modules, job and subsystems may be reviewed in reported outputs. It must be noted, however, that multiple (or repeated) inclusions of given plan level elements (e.g., a given module may be included in two different subsystems) will be processed in a special manner. That is, the data first collected for the given element will be re-utilized as input to required summarizations at superior levels in the report hierarchy. Output of the given report element itself will be made in association with its first occurrence only.

If, for example, a given subsystem element were included in two different system hierarchies, the subsystem hierarchy should be specified in connection with its first occurrence. Any redefinition of that subsystem hierarchy (in subsequent occurrences) will be noted (with advisory messages) and processing will continue as if such redefinition were not present. The occurrence of repeated inclusions of a given plan level element also causes advisory messages to be generated so that the user can take corrective action if the repeated inclusion is inadvertent.

It will be noted in Figure 3-09(a) that the PURGE Function is applicable to all collected/archived units. The actual names of the archived units are determined by the unique serial number and date of collection so that reference should be made to the MGMT section index (by using the INDEX Function) for a specific listing of archive unit names. The PURGE Function is not applicable to any other units in the MGMT section since both format units and management input units may be deleted via the MDFORMAT and MDUPDATE Functions, respectively. The management plan unit (having been added by the CREATE Function) is not deleted until the MGMT section is terminated.

MDCR Functions

When data has been collected, a management report is generated when an MDPRINT Function with the REPORT=MD option is requested. The management data report has a general format which provides appropriate identification of the level of data and the specific items for which values are printed. Figure 3-05 contains a sample management data report.

The management data collection capability consists of five special Functions as well as support from previously described Functions. This set of capabilities enables PSL users to design management data collections to reflect the needs of a specific project.

The special Functions which provide the data collection capability are described in the following paragraphs.

3.1.6.1 MDPLAN

The Management Data Plan Function is used to define the management data report structure. The plan unit (without data contents) is added automatically when a MGMT section is created. The MDPLAN Function is then used to modify the contents of the plan unit.

Example:
```
** CREATE PROJ=FHACD129,LIBR=NEWCODE,
**        SECTION=MGMT
** MDPLAN
(subfunction and source cards to modify the plan)
```

3.1.6.2 MDFORMAT

The Management Data Format Function is used to define the data items that may be reported at the five record levels previously described. Both automatically collected data items and manually input data items may be defined. Format units can be added, changed, or deleted by MDFORMAT as required.

Example:
```
** MDFORMAT PROJ=FHACD129,LIBR=NEWCODE,
**          OP=ADD,LEVEL=SYSTEM
(source cards)
```

3.1.6.3 MDUPDATE

The Management Data Update Function is used to maintain the manually provided data which resides in management data units. This data can be added, changed, or deleted as required.

```
Example:
** MDUPDATE PROJ=FHACD129,LIBR=NEWCODE,
**          UNIT=TIPTOP,OP=ADD,
**          LEVEL=SYSTEM
(source data cards)
```

3.1.6.4 MDXCHECK

The Management Data Exception Checking Function is used to annotate "excepted" data values in a management report. Exception check specifications are maintained in a management data unit corresponding to a denoted element in the report structure (i.e., plan unit). The specifications reference numeric data items contained in the associated record level format.

```
Example:
** MDXCHECK PROJ=FHACD129,LIBR=NEWCODE,
**          UNIT=TIPTOP
(subfunction and data cards)
```

3.1.6.5 MDCOLLECT

The Management Data Collection Function is used to collect and archive the data designated by MDPLAN, MDFORMAT, MDUPDATE, and MDXCHECK. Recycling of cyclic data accumulations and archiving data are options of the MDCOLLECT Function.

```
Example:
** MDCOLLECT PROJ=FHACD129,LIBR=NEWCODE,
**           ARCHIVE=YES,RECYCLE=NO
```

MDCR Applications

The example shown in Figure 3-09(c) represents a typical "off-the-cuff" applicatiou of MDCR Functions to obtain management statistics on newly developed code in the system project PSLPRG library. A management section is first created to accommodate units to be added and updated by the MDCR Functions. The MDCR-PLAN unit is updated to delineate the modules whose source code statistics are to be collected and reported. The PSL modules which constitute the newly developed code are aggregated under a subsystem element named NEWCODE and the project and library in which the source code statistics are maintained is designated. Management data format units are then added for the planned report levels. Module level format items are specified according to guidance given in Figure 3-13. Output labels are provided to fully describe the items being reported (beginning in column 25) and a reference to the associated data or special item statistic is made (in columns 5 through 8) as directed by Table 3-2. Subsystem level format items are specified with reference to numeric data items collected at the module level. The special item referenced (i.e., number of modules) is calculated during management data collection operations as invoked by the MDCOLLECT Function. Using the MDCR-PLAN unit and MDCR-FORMAT units as directive inputs, the MDCOLLECT Function operates to retrieve the required unit statistics and to summarize those statistics at the module and subsystem levels. The collected statistics are stored in the MDCR-COLLECTION unit for input to the Management Data (MD) Report invoked by the MDPRINT Function. The MD report prints out the collected statistics beginning with module level data and ending with a subsystem level summary.

The data collection and reporting initiated in the preceding example may be readily extended to include manual data by additionally defining such manually input items at the subsystem level, for example, and adding/updating a management data input unit through the MDUPDATE Function as follows:

```
** MDFORMAT PROJ=MYUMC,LIBR=MYLIB,LEVEL=SUBSYS
** I
 015 RPTA05MNAME        MODULE NAME
** MDUPDATE OP=ADD,LEVEL=SUBSYS,UNIT=NEWCODE
   MNAME=PFJBE
   MNAME=PCMLE
   MNAME=PPJVE
   MNAME=PRJVE
   MNAME=PPGLE
```

```
** PARAM PROJ=MYUMC,LIBR=MYLIB,SECT=MGMT
** CREATE FMS=(20,20)
** MDPLAN
** I A=0
   PROJ=SYSUMC,LIBR=PSLPRG
    SUBSYS=NEWCODE
      MODULE=PFJBE
      MODULE=PCMLE
      MODULE=PPJVE
      MODULE=PRJVE
      MODULE=PPGLE
** MDFORMAT OP=ADD,LEVEL=MODULE
 010A303                NAME OF ORIGINATOR
 020A301                TOP UNIT TYPE
 030A302                DATE TOP UNIT ORIGINATED
 040A304                TOP UNIT LANGUAGE
 050A305                DATE MODULE LAST UPDATED
 060$001                NUMBER OF UNITS
 070A101                TOTAL LINES OF SOURCE CODE
** MDFORMAT OP=ADD,LEVEL=SUBSYS
 010$001                NUMBER OF MODULES
 020M070                TOTAL LINES OF MODULE CODE
 030M070MAX             LARGEST MODULE LINES OF CODE
 040M060                TOTAL NUMBER OF UNITS
** MDCOLLECT
** MDPRINT REPORT=MD
```

Figure 3-09(c). MDCR Application Example

A "repeated" manual input item format specification is inserted into the previously added subsystem level format unit following item 010 (number of modules). A management data input unit named NEWCODE is added and updated with free-formatted source data specifying the item name and item value for each item updated. The subsystem level format is referenced to verify and edit the input item whose RPT specification permits multiple (or repeated) values to be provided and stored for the given item. An edit check for an alphabetic input of five characters, maximum is made. Each stored input value is further identified with a determined sequence number (shown in the subsequently output "source data" listing for that unit) for use in deleting or changing particular item values in subsequent manual-item updates.

If the source code being subject to management data collection and reporting were under development, it would be particularly appropriate to utilize the MDXCHECK Function for making automatic checks on the progress of that development. For example, the number of real units might be compared with the total number of units (i.e., real plus stub units) present in PSL storage. Since the subsystem format already contains a reference to the "total number of units", only a reference to the number of real units need be added. This would be done as follows:

```
** MDFORMAT PROJ=MYUMC,LIBR=MYLIB,LEVEL=MODULE
I065A202               NUMBER OF REAL UNITS
** MDFORMAT LEVEL=SUBSYS
I050M065   UREAL       NUMBER OF REAL UNITS
M040M060   UTOTAL      TOTAL NUMBER OF UNITS
```

The needed item must be inserted in the module level report format so that it may be referenced by and summarized at the subsystem level. The needed subsystem format item is inserted and a previously defined item is modified to provide an item name for mnemonic reference. An exception check specification may be provided immediately following the above such as follows:

```
** MDXCHECK UNIT=NEWCODE
** I
UREAL  UTOTAL:V20,110177,MANAGEMENT ACTION REQUIRED
```

This exception check is inserted into the management input unit named NEWCODE previously added through the MDUPDATE Function. The exception specification is verified against the subsystem level report format through correspondence with the item name entries. The exception check itself is performed when the MDCOLLECT Function is next invoked. The specification submitted is interpreted as follows.

If the number of real units is less than the total number of units witʰ a variance of twenty percent or more on or after November _, 1977, the message "management action required" will be given.

The message will follow the line which reports the number of real units and will specify the computed variance and identify the item number of the item to which the number of real units is being compared. Since each reported item is identified by item number as well as item name and output label, the reference to the compared item number may be readily related to the "total number of units" item in that same subsystem level report. The exception check may then be verified by an "on-the-spot" comparison of the two item values.

This concludes the example of MDCR Function utility in a typical management data collection and reporting application. Familiarity gained with the MDCR facility through initial application will confirm the PSL capability to collect and report management data statistics. The MDCR Functions may subsequently be applied to satisfy evolving management requirements by providing pertinent and timely information on the status and progress of software development activities.

3.2 Composition Guides

The PSL system is initialized in the user's run stream by the following card:

```
Col
1
@USE       PSL.,DMA*PSL.
@ADD       PSL.RUN
```

Following the @ADD card, one or more PSL Function cards are used to direct the PSL system to perform the desired processing. If user data cards are appropriate for a particular Function, they are inserted immediately after the associated Function card. Specific Function formats and examples of their use are given under the individual Functions in subsection 3.4. Examples of job input are shown in Appendix B.

The PSL Function cards are processed in the order in which they are encountered in the run stream (with the exception of the subfunctions under a CHANGE, DOCUMENT, MDPLAN, MDFORMAT, or MDUPDATE Function). Thus a user may prepare a library and use the library in the same activity. The end of the stream of PSL Function cards (and their accompanying data cards) and execution of the PSL system is indicated by the following cards:

```
Col
1
@END       PSL
@XQT       PSL.BCTL
```

The @ADD and @END are Exec 8 control cards and must follow the formats for such cards.

If Functions or options are used which require tape definiticns, the Exec 8 tape assign card with the file name UT is inserted after the @END card. Examples are given under the appropriate Functions.

The following subjects, which have general application over several PSL Functions, are discussed below:

a. High-level parameters
b. Multi-library search
c. Independent files
d. File Management Space
e. Spawned jobs
f. Stub generation
g. Error handling
h. INCLUDE statements
i. Name lengths and restrictions
j. Management data facility
k. Special-case card data

3.2.1 High-Level Parameters

The following parameters are cumulative during execution of the PSL system:

a.	PROJECT	Project name
b.	LIBRARY	Library name
c.	SECTION	Section name
d.	PASSWORD	Section password
e.	PGMR	Program name

Once the value for each of these keywords is established, either by a Function card or by derault, it remains in effect. All of these parameters may be replaced by another value by using the keyword on another Function card. A PARAM Function card is ordinarily used to establish these values, but the keywords may appear on any Function card for which they are valid. A section password is required to access a section if the password was assigned when the section was created and if the processing will modify the contents of the section.

3.2.2 Multi-Library Search

The following Functions allow an optional multi-library search during retrieval of input:

a. COMPILE
b. LINK
c. EXECUTE
d. MDCOLLECT
e. MDPRINT

A maximum of nine libraries may be searched.

The order of the search through the libraries is determined by the numerical digit in the following two special sets of keywords:

a. PROJ1, PROJ2, PROJ3, ..., PROJ9
b. LIB1, LIB2, LIB3, ..., LIB9.

Corresponding numeric digits are always paired, if they are declared, as:

PROJ1/LIB1,PROJ2/LIB2,etc.

It is not necessary to use all intermediate numerics. If specific numerics are skipped in one set and not skipped in the other, the preceding value for that set of keywords is used.

Example:
PROJ1=project-one, PROJ3=project-three
LIB1=library-one,LIB2=library-two,LIB3=library-three
These sets of keywords result in the following search sequence:

First	-	project-one/library-one
Second	-	project-one/library-two
Third	-	project-three/library-three

The keyword PROJ1 is synonomous with PROJECT, and the keyword LIB1 is synonomous with LIBRARY, in the five PSL Functions COMPILE, LINK, EXECUTE, MDCOLLECT and MDPRINT.

3.2.3 Independent Files*

The PSL system is designed to store card-image data in blocks in a random file. (See the discussion of the Structure of a Section in Appendix A.) This card-image data is read by special PSL access routines when it is retrieved by modules in the PSL system. However, if the data must sometimes be read by programs outside the PSL system, the data may be stored in a separate, or independent file. Examples of such data are unstructured code, which will be ready by a standard compiler, and test data, which will be read by a user's program.

When a unit is added to a section, the user may specify the unit-type, if the default type is not appropriate. (Default values are discussed under the ADD Function subsection 3.4.1). If the user adds a new unit (to the SOURCE section) for which the language is not a structured language supported by a

*The General Preprocessor can be used as an alternative method for storing unstructured source code. See Appendix I.

precompiler in the PSL system, the system will create an independent sequential file for the data. Since an unstructured unit is not processed by a precompiler and INCLUDE statements are not resolved, the independent SOURCE unit must be a complete compilable set of code and may not be reduced to smaller units, as may be done with structured code.

3.2.4 File Management Space (FMS) Parameter

The File Management Space Parameter allows the user to specify the size of files to be created by PSL. The PSL keyword "FMS" has two values. The first value is the number of tracks reserved and the second is the maximum number of tracks allowed. For the Project Index and Section files, with the exception of the PROGRAM section, the number of tracks reserved must equal the maximum number of tracks. The PROGRAM Section and Independent units can have different values for the reserved and maximum tracks. The standard defaults for the various files are given in Figure 3-10.

3.2.5 Spawned Jobs

Certain PSL Functions, called job spawning Functions, require the execution of independent programs such as precompilers, compilers or user written programs as part of their processing. To invoke an independent program, the PSL Function retrieves a pre-stored JCL procedure, modifies the JCL and writes the modified JCL on a temporary file. Subsequent Functions may append additional JCL on this file. Also, JCL may be added directly to this file by the user with the JCL Function. When the end of the PSL input is reached and all the PSL Functions have been processed, the temporary JCL file is extended (@ADD) onto the user's run.

The job spawning Functions are COMPILE, LINK, EXECUTE, MDCOLLECT, MDPRINT, and JCL. JCL procedures provided with the PSL system for use with these Functions are described in Appendix D. The EXECUTE Function invokes the users program using the JCL stored by the user in the JOB section of his library. User stored JCL is described in section 3.4.9 with the EXECUTE Function.

Function	File	File Mode	File Size in TRACKS (reserved, max.)
INITIAL	Project Index	Random	2,2
CREATE	Section	Random	10,10
CREATE	Project Section	Project File	10,10
ADD	Indep. unit only	Sequential	1,2

Figure 3-10. Default File Parameters

The user should not use the TERMINATE Function (to purge a section) in the same run with a job spawning Function which uses that section. The spawned job will not execute until after the other Functions, such as TERMINATE, have completed, and required section will no longer exist.

3.2.6 Stub Generation

The PSL generates stubs for missing units under two conditions:

a. When structured source code is added to a library, a dummy SOURCE unit (SOURCE stub) is provided for any unit w ich is referenced in an INCLUDE statement, but has not yet been added to the library. See description of the INCLUDE statement under subsection 3.2.8. Accounting information for the source stub is stored in the SOURCE section of the user's library.

b. When an object module is requested on an INCLUDE card in the user's collector control cards and the object module is not found by the PSL system, a dummy object module (OBJECT stub) is provided to the Collector by the system. A message will be printed on the system output file when the object stub is executed. The object stub is not in the user's library.

3.2.7 Error Handling

When an error is encountered on a Function card, an error message is generated and the remaining keywords for that Functions are scanned, but no library processing is done. Input is read and printed, but not processed, until the next Function card is found. If an error is found on a PARAM Function, no processing takes place until another PARAM Function card is found.

If an error occurs while processing a Function, the PSL system will attempt to undo whatever processing has taken place and will then terminate the Function. Advisory and error messages (see Appendix C) will be generated. Processing will resume at the next Function card.

It is recommended that a PARAM Function card be used to establish high-level parameters, to assure that intermediate Function cards will be skipped when one of these parameters is in error.

3.2.8 INCLUDE Statements

An INCLUDE statement is used in source code and with loader control cards to refer to a unit which will be retrieved and substituted in-line for the INCLUDE statement. The format of the INCLUDE statement is:

```
     numerics             INCLUDE              unit-name
```

Leading numerics are ignored. The word "INCLUDE" must be preceded by at least one space or must begin in column one. The unit-name must be preceded by at least one space and is terminated by a space, a period, a comma, a semi-colon, or a dollar-sign. Columns 73 through 80 are ignored. An INCLUDE statement may not be continued.

The INCLUDE statement may be used in the SOURCE, PDL, and LINK sections as follows:

a. SOURCE - When the INCLUDE statement is used as a line of code in a SOURCE unit, it refers to a lower-level unit of SOURCE code which will be substituted in-line for the INCLUDE statement. Such INCLUDE statements are resolved by a preprocessor before the code is passed to the compiler. Therefore, INCLUDE statements may only be used in modules which are written in SCOBOL, SJOVIAL or SPFORT, or modules which are processed by the General Preprocessor (see Appendix I). Independent units may not be INCLUDED by other units and should not contain INCLUDE statements. When a non-independent unit is added to a SOURCE section, or modified, the INCLUDE references are checked. If an included unit does not exist in the section, a SOURCE-stub unit is created and is tagged with the name and language of the including-unit. When the including-unit is compiled, the lower-level unit is a stub, an appropriate statement is inserted by the preprocessor to generate an output message when the code is executed. INCLUDE statements may be nested to a depth of 50 levels. The hierarchical pattern of nesting may be inserted by invoking the Program Structure (PS) report under the MDPRINT Function. If an error is detected during the INCLUDE processing, an appropriate message is printed and a "%" is inserted immediately before the word "INCLUDE" in the user's code.

b. PDL - The INCLUDE statement is processed in the PDL section in the same way as in the SOURCE section. A Program Structure report may be generated for a unit in the PDL section.

c. OBJECT - When the INCLUDE statement is used with collector control cards in the LINK section, it refers to a compiled unit (a module) in the OBJECT section for which an IN card will be inserted into the input stream passed to the Collector. If the unit is not found in the libraries selected on the LINK (or EXECUTE) Function card, an OBJECT-stub unit is substituted in its place (with the unit-name as the external-name of the stub). This unit appears as a "STUB" in the load map of the user's program and a message will be printed when the OBJECT stub is executed.

3.2.9 Name Lengths and Restrictions

The maximum lengths allowed for the various types of user-assigned names are tabulated in Figure 3-11.

3.2.10 Management Data Cycling and Archiving Operations

Data cycling is the periodic resetting of those management data items for which values have been accumulated over a period of time. Data cycling can be optionally defined to permit the user to specify the period for which values are to be accumulated before being reset.

Item	Maximum Length	Comments
Project	12	
Library	7	"SYSTEM", "PROJECT" may not be used.
Section	6	Standard names only.
Units		"ALL" may not be used.
SOURCE section		
MAIN CALLED INDEP	12	Top unit may become program ID.
INCLUDED	30	
LINK section	12	Name becomes LOAD entry and absolute element name.
Other sections		
INDEP	12	Stored as separate file.
Other types	30	

Figure 3-11. Name Lengths and Restrictions

Archiving enables the user to retain previously collected data for a historical review of project status.

3.2.10.1 Cyclic Data Operations

"Cyclic" management data is first implemented in the Unit Accounting Record for PSL sections in which the MGMT=YES option is indicated. Two cyclic data items are maintained in that record: lines input per cycle and number of updates per cycle. A project is given the option of defining the duration of the unit level cycle thrugh one of the following methods:

a. Unit level cycle - If a format unit for unit level data is established, the cycle duration for cyclic items in the unit accounting record can be specified.

b. Module level cycle - In lieu of establishing a format unit for unit level data, the cycle duration established in the format unit for module level data will be applied to the unit accounting record.

The determination as to when the specified cycle period has elapsed is made when a management data collection is performed. If the number of days specified for the cycle period is equal to or exceeds the number of days elapsed since the accumulation of cycle statistics was begun, the cyclic data item contents will be reset to zero (i.e., recycled) after the collection of that data. A new data cycle is started and cycle duration is determined from that date for all SOURCE section units linked through the management plan; that is, through the module names listed in the management plan unit.

Cyclic data items, as described in the module, job, subsystem, and system level formats, are defined through the MDFORMAT Function (paragraph 3.4.15), while the unit accounting record items are pre-defined by the PSL system.

Cyclic data items are specified in a given level format unit to compute the cumulative changes in value of other items in that same format. For example, if "total-lines-added" is defined as an item at the module level (derived from a summation of the equivalent item in the included unit's accounting record), a cyclic data item may be defined to compute the number of lines added during the period of time defined by the cycle. The defined "lines-added-per-cycle" item will reference the "total-lines-added" item with the indication that a cyclic accumulation is required.

The ability to define cyclic data items extends to the system level format which represents the top level of data collection and summarization for a project. It is suggested that the cycle duration for the system level record might be greater than for the subsystem level and that, in turn, might be greater than for the module level record. For example, the system level data may be cycled on a monthly basis, the subsystem on a bi-weekly basis, and the module on a weekly basis.

Archiving of collected data might be performed for system level data only at the end of the system level cycle period. Cyclic data item values in the archived system level record would thereby represent the change in value of referenced items as has occurred during the cycle just concluded.

3.2.10.2 Archiving Data

Automatic archiving is generally associated with the conclusion of the system level cycle period. However, an option to archive other than system level data is provided to permit any or all levels to be archived. The determination to automatically archive is made when the data is collected, based upon the current date, the start-of-cycle date, and the cycle duration. Actual archiving is not performed, however, until the next data collection remains the "current" data collection until such time as it is archived and/or replaced.

Although automatic cycling and archiving may be flexibly defined and used to satisfy most user requirements, a manual cycling and archiving capability is provided to ensure that special user requirements are accommodated (paragraph 3.4.14, MDCOLLECT). The options to manually suppress or enable cycling and/or archiving are available for each data collection performed. This capability may be used in combination with the automatic cycling and archiving. It is also possible to suppress the automatic cycling option so that only manual cycling and/or archiving is performed.

Each archived unit is uniquely identified at the time of archiving by affixing a suffix to its basic name. This suffix consists of a three digit serial number and a six character date. The assigned serial number (starting with 001) is increased by one for each subsequent collection archived. The six character date, given in MMDDYY format, signifies the day on which the data collection was performed.

The PSL INDEX Function can be used periodically to obtain a list of the MGMT section units. A scan of this listing will give a very quick indication of the specific dates on which archived data was collected and, in reviewing the serial number designations, determine whether any archived collections are missing. The ordering of archived collection units by serial number allows for a rapid determination of the elapsed days between successive archives.

While it is of considerable convenience to retain the more recently archived collections in the Project MGMT section, each such collection takes up section storage space. Two options are open to the user when the remaining free space drops below a determined threshold:

a. TERMINATE, CREATE, and RESTORE - the section can be terminated using the TERMINATE Function with the BACKUP option, recreating the MGMT section with a larger space allocation using the CREATE Function, and restoring the backup file using the RESTORE Function.

b. BACKUP and PURGE - the section can be backed-up using the BACKUP Function and archived collections can be selectively deleted using the PURGE Function.

The latter method could be used to remove the older archived collections from the MGMT section. The backup tape produced in that operation should be retained and labeled to provide for the restoration of purged archive units when and if required.

The procedures to be followed for administering and controlling backup and archived material are a prerogative of the user or the site. Currently, the PSL facility is designed to provide a practical means for storing and selectively retrieving archived data prerequisite to the production of management data history reports.

3.2.11 Special-Case Card Data

The user may wish to enter PSL Function or Subfunction cards as data. A special handling facility is provided to enable cards with "**b̸" in the first three columns to be maintained in a card-image data unit. The addition, replacement, and insertion of this special-case card data is accomplished by enclosing the data within a PSL DATA/PSL ENDATA card pair.

The format of the PSL DATA card and the PSL ENDATA card follows:

```
**   DATA initial-character
     (data cards)
**   ENDATA closing-character
```

The initial-character must match the closing-character to mark the end of the data stream.

The following example shows how a user may maintain units containing special-case card data:

```
**   PARAM PROJ=UMC1234,LIBRARY=NEWCODE,SECTION=TEST
**   ADD UNIT=NEWTEST,UTYPE=MAIN
**   DATA A
**   PARAM PROJ=UMC1234,LIBRARY=NEWCODE,SECTION=SOURCE
**   CHANGE UNIT=TIPTOP
**   D L=(1,10)
**   M L=14,C=12,T=NEW
**   COMPILE UNIT=TIPTOP,PROCEDURE=COBOL
**   ENDATA A
**   CHANGE UNIT=NEWTEST
**   M L=1
**   DATA A
**   PARAM PROJ=UMC456,LIBRARY=JOHN,SECTION=SOURCE
**   ENDATA A
```

3.3 Conventions and Glossary

3.3.1 Format Conventions

The following conventions will be employed in formats and examples:

a. PSL Function card parameters are shown in upper case if the parameters are to be entered exactly as shown.

b. PSL Function card parameters are shown in lower case if the parameters are to be replaced by the user with appropriate information.

c. Square brackets [] indicate that the enclosed item is optional and may be omitted.

d. Braces { } indicate that one of the enclosed values should be used.

e. A literal string is a string of characters (in the Sperry Univac 1100 series character set) bounded by quotation marks.

f. When more than one value choice is indicated for a keyword-value-entry and one of the values is underlined, the underlined value will be provided as the default value if the keyword is omitted.

g. The ellipsis ... indicates that the preceding item may be repeated.

3.3.2 Glossary

The following terms are used in this document with the associated specific meanings:

a. A "Function" is one of 28 PSL operations which is invoked with a PSL card as defined in subsection 3.4. The valid PSL Functions are shown in Figure 3-12.

Functions

ADD
AUTHOR
BACKUP
CHANGE
COMPILE
CREATE
CSCAN
DOCUMENT
EXECUTE
INDEX
INITIAL
JCL
LINK
MDCOLLECT
MDFORMAT
MDPLAN
MDPRINT
MDUPDATE
MDXCHECK
MOVE
PARAM
PERFORM
PRECOMPILE
PURGE
REPLACE
RESTORE
SOURCE
TERMINATE

Subfunctions

COPY
DELETE
EJECT
HEADER
INSERT
MODIFY
SHIFT
SPACE
TEXT

Keywords

PSL

AFTER
ALL
ARCHIVE
BACKUP
CALL

COLUMN
COMPRESS
CYCLE
ELEMENT
FILE1-9
FMS
FROM
HISTORY
HPRINT
HSPACE
INDENT
INPOOL
JOB
KEY
LADJUST
LANGUAGE
LEVEL
LIB1
.
.
.
LIB9
LIBRARY
LINE (S)
LINK
LOAD

LSPACE
MEDIA
MGMTDATA
MOD
MODULE
NBRLINES
NEST
OBJECT
OLDLIB
OLDPROJ
OLDSEC
OLDUNIT

OP
OPGMR
OPTION
OUTPOOL
PAGE

PASSWORD
PGMR
PLINES
PROCEDURE
PROJ1
.
.
.
PROJ9
PROJECT
RECYCLE
REPLACE
REPORT
SECTION
SPCHECK
SPLENGTH
STANDARD
START
STRING
SYSTEM
SUBSYSTEM
TO
UNIT

UPGMR
USPACE
UTYPE

Figure 3-12. Functions, Subfunctions, and Keywords

b. A "Subfunction" is one of nine PSL directives which are used only after a CHANGE or DOCUMENT Function card. They are COPY, DELETE, EJECT, HEADER, INSERT, MODIFY, SHIFT, and TEXT.

c. A "keyword" is a word-symbol used to identify a parameter on a PSL Function card. The first four letters of keywords (except PROJ2...PROJ9, FILE1...FILE9) must be spelled exactly as given in Figure 3-12 or the PSL system will not recognize the keyword.

d. A "module" is a compilable set of code. It may be retrieved from one or more units in the Source section. Compilation will result in one unit in the OBJECT section. A module is not necessarily executable independently.

e. A "program" is an independently executable set of code. The code may consist of one, or more than one, relocatable element from a PROJECT section. Linking a program will result in one absolute element in the PROJECT section, which may be executed, and an appropriate record in the LOAD section.

f. The special characters ",", "/", and "=" are required if they appear in the format of a Function card.

3.4 Input Format of Function Cards

The general format of a PSL Function card is:

```
col
1
** function  keyword=value-entry,keyword=value-entry ...

where value-entry may be:
     a.  value
     b.  (value,value)
```

The presence of "** " in columns 1 through 3 indicates that card is a PSL Function card or a PSL continuation card. The name of the Function may begin in any column after column three, but it must terminate before column 73. Columns 73 through 80 ar^ ignored by the PSL system. The keyword-value-entries follow the Function name, and the first keyword must be preceded by one or more blanks. After the beginning of the first keyword, a blank ends the scan of a PSL Function card. The remainder of the card is ignored. The keyword-value-entries may be interrupted after any comma (unless the comma is inside quotes) and continued on a PSL continuation card. The data on a continuation card may begin in any column after column three and may be continued through column 72. A subsequent blank will terminate the scan. If keyword-value-entries are present, at least one keyword must appear on the same card as the Function. The valid PSL keywords are listed in Figure 3-12. All PSL keywords, except PROJ2...PROJ9 and FILE1...FILE9, may be abbreviated to four characters. In addition, the Subfunctions under the CHANGE and MDPLAN Functions, and the associated Subfunction keywords, may be abbreviated to one character.

```
Example:
** PARM PROJ=FHACD147,LIBR=PROVEN,SECT=SOURCE
** CHANGE UNIT=TIPTOP
** M L=(10,12)
(new source data cards)
** D L=14
** ADD SECT=JOB,UNIT=TIPTOP
(JCL for user's execution)
```

If a keyword value contains one of the special characters "/", ")", "(", ",", quote, or space, the value must be enclosed in quotation marks. Within quotation marks, the occurrence of two quotation marks in sequence are interpreted as a single quotation mark and part of the value.

The detailed specifications for the keywords and value for each PSL Function, with explanatory information, are given on the following pages. Multiple keyword/value pairs may be grouped on a single PSL Function card.

The user may insert source data cards by means of an @ADD,D card.

3.4.1 ADD

The ADD Function is used for the original introduction of data into a unit. The accounting information for the unit is initialized by the ADD Function. The actual data cards for the unit follow the ADD Function card in the run stream.

```
 **  ADD   PROJECT=project-name,
 **        LIBRARY=library-name,
 **        SECTION=section-name,
 **        PASSWORD=section-password,
 **        UNIT=unit-name,
 **        LANGUAGE=language-name,
[**        KEY=unit-key,]
[**        UTYPE=unit-type,]
[**        FMS=fms-entry,]
[**        PGMR=programmer-name]
```

The values for the ADD parameters are:

project-name - name of project.

library-name - name of library.

section-name - name of section; must be SOURCE, PDL, JOB, LINK, TEST, or TEXT.

section-password - password which was assigned when the section was created; not required if a password was not assigned.

unit-name - name of unit to be added; maximum length of 30 characters; maximum length of 12 characters if unit-type is INDEPENDENT or if unit is being added to LINK section; maximum length of 12 characters if unit is a top unit (unit-type is MAIN, CALLED, INDEPENDENT) in SOURCE section; ALL is a reserved word and cannot be used for a unit-name.

language-name - the name of the programming language for the unit; required only in SOURCE section for a top unit (MAIN, CALLED, INDEPENDENT) and for any included unit for which a stub does not already exist; language-name is not checked for validity, but COMPILE Function is dependent on language-name; maximum of eight characters. Units with language SPFORT, SJOVIAL or SCOBOL will be automatically indented when printed.

unit-key - the key that will be required in order to modify or purge the unit; maximum of twelve characters; keys are not checked for read-only Functions.

unit-type - one of the word-symbols (MAIN, CALLED, INCLUDED, INDEPENDENT) which describes the structural position of the unit; may be entered in four-character abbreviated form (MAIN, CALL, INCL, INDE); usually assigned from default values (see below); INCLUDED and CALLED are valid for SOURCE and PDL sections only; MAIN and CALLED units are processed the same, but the distinction is used in Management Data reports.

fms-entry - used to enter FMS parameters for file; if INDEPENDENT unit is being added; default size for INDEPENDENT file is 1 reserved track, 2 maximum tracks.

programmer-name - name to be placed in accounting record; maximum of twelve characters; default is project identification for job.

A unit-type of INCLUDED is not normally declared by the user. In top-down development, INCLUDEd units are automatically generated in the library as stubs with a unit-type of INCLUDED. Code may be added to the stub unit with the ADD Function or with the REPLACE Function, and there is no need to declare the unit-type or the language. However, if a unit is to be an INCLUDEd unit and it is added to a library before a higher-level unit has caused a stub to be generated in that library, both the unit-type (INCLUDED) and the language must be explicity declared.

If a user attempts to ADD code to a unit which exists in that library and which is not a stub, the ADD Function will be rejected. When a unit is ADDed to a section, and no stub exists, and the unit-type is not provided as a parameter, the following default values are used:

Section	Unit-Type
SOURCE[1]	
Structured language (SCOBOL, SPFORT, SJOVIAL)	MAIN
Unstructured language (ASM, COBOL, FORTRAN, JOVIAL)	INDEPENDENT
PDL	MAIN
JOB	MAIN
LINK	MAIN
TEST	MAIN
TEXT	MAIN

The following example builds the top unit of a module and adds code to one of the generated stubs:

```
** PARAM PROJ=FHACD129,LIBR=NEWCODE,SECTION=SOURCE,
**      PGMR=BERT
** ADD UNIT=TIPTOP,LANG=SCOBOL
(source cards for TIPTOP main unit)
** ADD UNIT=TIPTOP-DATA-DIVISION
(source cards for TIPTOP-DATA-DIVISION included unit)
```

The following example adds an entire unstructured module:

```
** ADD PROJ=FHACD147,LIBR=UNSTRUC,SECTION=SOURCE,
**     UNIT=FTPROG,LANG=FORTRAN,FMS=(10,25)
(source cards for FTPROG complete module)
```

[1] Also see Appendix I.

3.4.2 AUTHOR

The AUTHOR Function is used for the printing of a list of unit names (OPTION=INDEX), or listings of the actual units (OPTION=SOURCE), which were originally generated by (PGMR=SMITH) and/or updated by (UPGMR=SMITH) a specific programmer. The OPTION=SOURCE can only be used for units which are in card-image format; that is, units in the SOURCE, PDL, LINK, JOB, MGMT, TEXT, and TEST sections.

```
**   AUTHOR  PROJECT=project-name,
**           LIBRARY=library-name,
**           SECTION=section-name,
**          {OPTION=   SOURCE }
            {          INDEX  },
**          {OPGMR=originating-programmer,} only one required
**          {UPGMR=update-programmer      }
```

The values for the AUTHOR parameter are:

project-name - name of project.

library-name - name of library.

section-name - name of section; if OPTION=SOURCE, must be SOURCE, PDL, LINK, JOB, MGMT, TEST, or TEXT.

option-source - required if source listing of units are desired; the default value is INDEX, which lists only the unit name and associated information; the INDEX option applied whenever the OPTION keyword is omitted.

originating-programmer - the name of the programmer who originally added the unit to a section.

update-programmer - the name of the programmer who last updated the unit.

One of the keywords, OPGMR or UPGMR, is required for the AUTHOR Function. Both may be used only if the values of originating-programmer and update-programmer are the same. If OPGMR is used alone, the listing will contain all units which were originally added to the section by a specific programmer. If the update-programmer is different from the originating-programmer, the name of the update-programmer will also be printed on the index listing. If UPGMR is used alone, the listing will contain all units in the section which were last updated by a specific programmer. If the originating-programmer name is different from the update-programmer name, the originating-programmer name is also printed on the index listing.

The following example:

a. prints a source listing of all units originated by a programmer named STARTER,

b. prints an index listing of all units which were last updated by a programmer named FIXER, and

c. prints an index listing of all units which were originated and/or last updated by a programmer named SUPER.

```
**  PARAM       PROJ=FHACD129,LIBR=NEWCODE,SECT=SOURCE
**  AUTHOR      OPTION=SOURCE,PGMR=STARTER
**  AUTHOR      UPGMR=FIXER
**  AUTHOR      PGMR=SUPER,UPGMR=SUPER
```

3.4.3 BACKUP

The BACKUP Function is used to save sections of a library on tape. The BACKUP may be invoked for a complete project, a library, or a section.

```
**   BACKUP     PROJECT=project-name,
**              LIBRARY= {library-name} ,
                         {ALL         }
**              SECTION= {section-name}
                         {ALL         }
```

plus a tape card with a file name of UT, as follows:

```
Col
1
@ASG,C  UT,T
```

This is an Exec 8 control card and follows the appropriate format. The tape card must be placed after the @END PSL card at the end of the PSL Function cards.

The values for the BACKUP parameters are:

project-name - the name of the project; must be the same as the project identification on the @RUN control card for the job.

library-name - the name of the library in which the section is found; ALL indicates that all libraries in the project are to be saved.

section-name - the name of the section to be saved; ALL indicates that all sections in the library are to be saved; the section-name is not required and is ignored if LIBRARY=ALL.

The following example will invoke the PSL system and produce a BACKUP tape:

```
Col
1
@USE        PSL.,DMA*PSL.
@ADD        PSL.RUN
**BACKUP    PROJ=FHACD129,LIBR=NEWCODE,SECT=ALL
@END        PSL
@ASG,CL     UT,U9V,ZØ162Ø
@XQT        PSL.BCTL
```

Between the @ADD card and the @END card, the user may place an entire series of PSL Functions, but the user should be aware of the sequence in which the output tape will be used. The tape, if present, is opened once at the beginning of the job and closed at the end. When a Function is invoked which uses this tape, the output from that Function is written on the tape without rewinding the tape. Thus a user may select multiple BACKUP Functions and the output will be arranged sequentially. A TERMINATE Function, with BACKUP=YES, may also be inserted into this pattern, if desired by the user. However, the user must be careful not to use the SOURCE Function, with MEDIA=TAPE, in the same job as a BACKUP or a TERMINATE-with-BACKUP, since the resulting output would also be written sequentially to the same tape.

The BACKUP Function tries to obtain FMS parameter information for each independent unit file in the section being backed-up. If available, the FMS parameters are written on the backup tape with the file control for use in recreating the file when the sections is subsequently restored. If, however, the independent file was created by a user other than the owner of the project (project-name=project identification), the BACKUP Function cannot obtain the FMS parameters, and only the file content is written on the backup tape. In this case, the user can provide new FMS parameters for the RESTORE Function or the RESTORE will use the PSL default values listed in Figure 3-10.

3.4.4 CHANGE

The CHANGE Function is used to modify the contents of a unit. Modification details are provided on Subfunction cards (COPY, DELETE, INSERT, MODIFY, and SHIFT) which follow the CHANGE card. New source statements follow the INSERT and MODIFY Subfunction cards.

```
 **  CHANGE      PROJECT=project-name,
 **              LIBRARY=library-name,
 **              SECTION=section-name,
 **              PASSWORD=section-password,
 **              UNIT=unit-name,
 **              KEY=unit-key,
[**              FMS=fms-entry,]
[**              LANGUAGE=new-language-name,]
[**              MOD=modification-level,]
[**              PGMR=programmer-name]
```

The values for the CHANGE parameters are:

project-name - name of project.

library-name - name of library.

section-name - name of section; must be SOURCE, PDL, JOB, LINK, TEST or TEXT.

section-password - password which was assigned when the section was created; not required if a password was not assigned.

unit-name - name of unit to be changed.

unit-key - key which was assigned to unit when it was added to section; not required if a key was not assigned.

fms-entry - used to change FMS parameters for file if unit was an INDEPENDENT unit.

new-language-name - new name to replace current value of language-name in unit.

modification-level - number to be checked against modification level in unit accounting record; a listing of the unit is provided, but no update takes place if values do not match; maximum value of 999.

programmer-name - name to be placed in accounting record; maximum of 12 characters; default is userid for job.

The Subfunctions which are used to CHANGE the unit are:

```
**  MODIFY LINES=  ┌line-nbr                        ┐
                   │(first-line-nbr,last-line-nbr)  │ ,
                   └ALL                             ┘
(new data cards)

        or

**  MODIFY LINES=  ┌line-nbr                        ┐
                   │(first-line-nbr,last-line-nbr)  │ ,
                   └ALL                             ┘
**          FROM=existing-character-string,
**          TO=new-character-string

        or

**  MODIFY LINES=  ┌line-nbr                        ┐
                   │(first-line-nbr,last-line-nbr)  │ ,
                   └ALL                             ┘
**          COLUMN=start-column-nbr,
**          TO=new-character-string
```

The values for the MODIFY Subfunction are:

existing-character-string - character string to be modified; character string must be enclosed in quotes if special characters* are embedded; maximum of 40 characters

Special character (commas, quotes, equal-sign, blanks, parenthesis and slashes) new-character-string - replaced existing character string on source card; character string must be enclosed in quotes if special characters are embedded; maximum of 40 characters. If length of a new string exceeds the length of the old string, characters to the right of the old string will be shifted right with truncation after column 80.

start-column-nbr - start position on source card where first-character of new string will be placed; maximum column number is 80. Characters existing on the line will be replaced.

```
**  DELETE LINES=  {line-nbr                          }
                   {(first-line-nbr,last-line-nbr)    } ,
                   {ALL                               }

**  SHIFT LINE=    {line-nbr                          }
                   {(first-line-nbr,last-line-nbr)    } ,
                   {ALL                               }

**        COLUMN=shift-indicator
```

The value for the SHIFT Subfunction is:

shift-indicator - shift-indicator consists of a character code (direction value "R" for right and "L" for left) followed by shift-nbr which is the number of positions to be shifted; maximum value is 80.

```
**  INSERT AFTER=  {line-nbr}
                   {ALL     }
(new data cards)

**  COPY AFTER=    {line-nbr}
                   {ALL     }
[**       OLDUNIT=old-unit-name,]
 **       FROM=  {from-line-nbrs                                }
                 {(first from-line-nbr,last from-line-nbr)      } ,
                 {ALL                                           }
[**       OLDPROJ=old-project-name,]
[**       OLDLIB=old-library-name, ]
[**       OLDSECT=old-section-name ]
```

*Special character (commas, quotes, equal-sign, blanks, parenthesis and slashe3)

The values for the COPY Subfunction are:

old-unit-name - name of unit from which source statements are taken.

old-project-name - name of project where old unit is located; defaults to current project.

old-library-name - name of library where old unit is located; defaults to current library.

old-section-name - name of section which contains old unit; must be SOURCE, PDL, JOB, LINK, MGMT, TEST, or TEXT; defaults to current section.

from-line-nbrs - line number or range of lines of old unit to be copied.

The Subfunctions need not be in any particular order. They will be sorted on beginning-line-number before they are processed by the CHANGE Function. Input order is preserved for equal beginning-line-numbers. All Subfunctions and their associated keywords may be abbreviated to four characters or to one character.

A value of zero is valid for line-nbr or first-line-nbr. If the value of last-line-nbr exceeds the highest line number in the unit, an error message will be generated, but the CHANGE will process the unit up through the last actual line. The line numbers used on the Subfunction cards should be those from the last listing of the unit. New line numbers will not be established for lines in the unit until the CHANGE Function has completed processing.

The user of INSERT AFTER=ALL will cause code to be appended after the last line of the unit.

A single line may not be referenced, during one CHANGE Function, more than six times, either directly or by inclusion in a range. Excess references will not be processed.

The CHANGE Function updates a unit by processing a line at a time and creating a new copy of the unit. If the unit is INDEPENDENT, a temporary file is used. If the unit is not INDEPENDENT, space must be available in the section file for a temporary copy of the unit blocks or the CHANGE will not be able to complete processing.

After the CHANGE processing has terminated, a listing of the unit is automatically generated showing the actual contents of the unit in the library.

The following examples show how a user may change the contents of a unit:

```
** PARAM  PROJ=UMC1234,LIBR=NEWCODE,SECTION=SOURCE
** CHANGE UNIT=TIPTOP,KEY=MYSTERY
** C  A=1,OLDU=ADUM,OLDLIB=PROVEN,FROM=1
** M  L=(5,9)
(source cards to replace lines 5-9;
  need not be the same number of cards)
** D  L=(10,15)
** I  A=23
(source cards to go between line 23 and line 24)
** S  L=24,COLUMN=R10
** M  L=26,TO="MOVE TO",FROM=MOVETO
```

The same change would be effected by:

```
** PARAM  PROJ=UMC1234,LIBR=NEWCODE,SECTION=SOURCE
** CHANGE UNIT=TIPTOP,KEY=MYSTERY
** I  A=1
(source card inserted after line 1)
** I  A=23
(source cards to go between line 23 and line 24)
** D  L=(5,15)
** I  A=4
(source cards to replace line 5-9)
** S  L=24,COLUMN=R5
** M  L=26,TO="MOVE ",FROM=MOVE
** S  L=24,COLUMN=R5
```

For the CHANGE of an INDEPENDENT unit, if the space assigned to the independent unit file is insufficient to contain all of the change unit, the PSL program will abort. In this case, the user may use the RESTORE Function to restore the unit from a previous BACKUP tape to its status before the CHANGE and rerun the CHANGE with the FMS keyword to increase the size of the independent file. Alternately, the SOURCE Function may be used to obtain a listing of the current contents of the change unit. Then, a CHANGE may be used to re-enter any source lines which were truncated with the FMS keyword to increase the size of the independent file.

3.4.5 COMPILE*

The COMPILE Function retrieves units from the SOURCE section, invokes the appropriate precompiler if the source code is structured, compiles the resulting stream of code, records the unit in the OBJECT section, and stores the compiled product as a relocatable element in the PROGRAM section. The unit name is the name of the top-most source unit of the module which is to be compiled. This name will be used for the name of the compiled OBJECT unit. The processing which is invoked by the COMPILE Function depends upon the language of the unit. Under the present PSL system, procedures exist to process languages ASM, ASMG (Compacted ASM), COBOL, COBOLG (Compacted COBOL), SCOBOL (Structured COBOL), FORTRAN, FORTRANG (Compacted FORTRAN), SPFORT (Structured FORTRAN), JOVIAL, JOVIALG (Compacted JOVIAL), and SJOVIAL (Structured JOVIAL). Procedure to compile additional languages may be added to the system. Procedures are automatically invoked by the language-name, or may be specifically invoked with the PROCEDURE keyword.

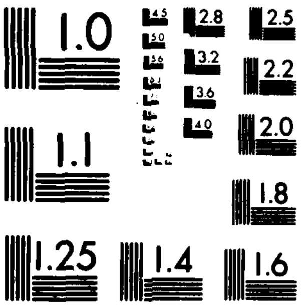
```
 ** COMPILE PROJECT=project-name,
 **         LIBRARY=library-name,
 **         UNIT=unit-name,
 **         PASSWORD=object-section-password,
[**         OBJECT=  {YES} ,  ]
[                    {NO }    ]
[**         PROCEDURE=special-compile-procedure,]
[**         {INPOOL=   {compool                      } } ]
[           {          {(compool-1,...,compool-9)    } } ]
[           {OUTPOOL=  {YES}                           } ]
[           {          {NO }                           } ]
[**         PROJ1=first-project-to-search,]
[**         PROJ2=second-project-to-search,]
                 .
                 .
                 .
[**         PROJ9=ninth-library-to-search,]
[**         LIB1=first-library-to-search,]
[**         LIB2=second-library-to-search,]
                 .
                 .
                 .
[**         LIB9=ninth-library-to-search]
```

*Also see Appendix I.

AD-A107 249   IBM FEDERAL SYSTEMS DIV GAITHERSBURG MD   F/G 9/2
PROGRAMMING SUPPORT LIBRARY (PSL). USERS MANUAL.(U)
MAY 78   F30602-77-C-0249

UNCLASSIFIED   DOD/DF-81/016A   NL

2 of 3



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

The values for the COMPILE parameters are:

project-name - name of first project to search for SOURCE units and name of project to be used for resulting OBJECT unit, if OBJECT=YES; value for PROJ1.

library-name - name of first library to search for SOURCE units and name of library to be used for resulting OBJECT unit, if OBJECT=YES; equivalent to value of LIB1.

unit-name - name of top-most SOURCE unit to be compiled; becomes name of OBJECT unit, if created; unit must be MAIN, CALLED, or INDEPENDENT unit-type; maximum length of name is twelve characters.

object-section-password - password of OBJECT section; required if a password was assigned to the section and if OBJECT=YES.

OBJECT=YES - resulting OBJECT module is recorded in OBJECT section and stored in PROGRAM section; this is the default value; if OBJECT=NO, compiled module is not available for use.

special-compile procedure - the name to be used to invoke a special JCL procedure to compile the module, instead of the name of the language associated with the SOURCE code; this procedure must have been previously stored in the system.

compool - the name of a previously assembled compool symbol table(s) (JOVIAL compiler ONLY).

OUTPOOL=YES - two modules are stored in the OBJECT section (COMPOOL symbol table, resulting OBJECT module); if OUTPOOL=NO only the OBJECT module is stored (JOVIAL compiler ONLY).

first-project-to-search - equivalent to value for PROJECT; if both are entered, the last one will be used. Also name of project where object unit will be stored if OBJECT=YES.

second-project-to-search - see subsection 3.2.2 for a discussion of a multi-library search.

.

.

ninth-project-to-search

first-library-to-search - equivalent to value for LIBRARY; if both are entered, the last one will be used. Also name of library when object unit will be stored if OBJECT=YES.

second-library-to-search - see subsection 3.2.2 for a discussion of a multi-library search.

.

.

.

ninth-library-to-search

The following example invokes a series of COMPILEs for JOVIAL units:

```
** PARAM       PROJ=FHACD129,LIBR=NEWCODE
** COMPILE     UNIT=POOLA,OUTPOOL=YES
** COMPILE     UNIT=POOLB,OUTPOOL=YES
** COMPILE     UNIT=MAINJV,INPOOL=(POOLA,POOLB)
```

Units POOLA and POOLB contain JOVIAL data declarations. The specifying of OUTPOOL=YES results in the generation of a COMPOOL symbol table for each unit. When the unit MAINJV is compiled, the previously assembled symbol tables are made available to the JOVIAL compiler for searching.

The following example will invoke a compile of a module using a multi-library search:

```
     ** PARAM        PROJ=FHACD128,LIBR=NEWCODE
     ** COMPILE      UNIT=TIPTOP,
     **              PROJ2=FHACD147,LIB2=PROVEN
```

Since TIPTOP is written in Structured COBOL (SCOBOL), the SCOBOL precompiler will process the units, starting with TIPTOP and working down through all INCLUDE statements. As each INCLUDE statement is picked up, the libraries are searched for the named unit. NEWCODE is searched first and then PROVEN. As each INCLUDEd unit is found, the code is read and added to the total stream of code which will be processed. Nested INCLUDEs are resolved in place so that the final module is in proper logical order. Comments are placed in the listing to shown the project and library where each unit was found. If a unit is a SOURCE stub, a DISPLAY statement is inserted in its place. Original unit-line-numbers are placed in columns 1 through 6 of the compiler input, for programmer reference.

The procedures which are provided with the PSL system are listed in Appendix D. See subsection 3.2.4 for a discussion about spawned jobs.

3.4.6 CREATE

After a project is initialized, the CREATE Function is used to build sections in a user's library. Section options are selected at this time. For a MGMT section, the management plan unit is initialized. A detailed description of the structure of a section is given in Appendix A.

```
 **  CREATE  PROJECT=project-name,
 **          LIBRARY=library-name,
 **          SECTION=section-name,
[**          PASSWORD=section-password,]
[**          FMS=fms-entry,]

  Section options:

[**          COMPRESS=  NO   ,]
[                       YES   ]
[**          MGMTDATA=  NO   ,]
[                       YES   ]
[**          SPCHECK=   NO   ,]
[                       YES   ]
[**          SPLENGTH=  50            ,]
[                       nbr-of-lines   ]
[**          KEY=plan-unit-key]
```

The values for the CREATE Function are:

project-name - name of project under which the section will be built.

library-name - name used to uniquely identify a group of standard sections under a project; maximum length of seven characters; one of each of the standard sections may be built under a given library.

section-name - name of section; must be one of the standard section names (JOB, LINK, LOAD, MGMT, OBJECT, PDL, SOURCE, TEST, TEXT, USER, PROGRAM).

section-password - password which will be required in order to write into the section; maximum of twelve characters.

fms-entry - used to enter FMS parameters for section file; default size is ten TRACKS reserved and maximum.

The options which may be selected for a section are described below:

a. COMPRESS - This option is appropriate only for the JOB, LINK, MGMT, PDL, and SOURCE sections. If option is selected, leading blanks are omitted on the file and reinserted upon retrieval. Trailing blanks are always suppressed. INDEPENDENT units are never compressed, regardless of the value selected for this option.

b. MGMTDATA - This option is appropriate for all sections. See Figure 3-01 for a detailed list of the accounting data which may be accumulated for a unit in these sections. The items which are listed in the Accounting Record are maintained whether or not this option is selected. The items in the Extended Accounting Record are only maintained if MGMTDATA=YES.

c. SPCHECK - This option is appropriate for the PDL and SOURCE sections only. If the option is selected, and if the language of a unit is supported by a structured-programming unit-print facility, the code of a unit will be checked during printing for adherence to structured-programming principles. Violations will be flagged. At present, only SCOBOL. SJOVIAL and SPFORT are supported for this option.

d. SPLENGTH - This option is subordinate to the SPCHECK option. The value of this option is used during a structured-programming check as the maximum number of lines per page. The default value is 50. Excess lines are flagged.

e. plan-unit-key - Used only when creating a MGMT section; the plan-unit-key, if specified, will subsequently be required to update the management data plan unit with the MDPLAN Function.

3.4.7 CSCAN

The CSCAN Function is used to scan for a specific string of data in any unit of a section. It will list the unit name and corresponding lines of code for each occurrence.

```
** CSCAN PROJ=project-name,
**       LIBRARY=library-name,
**       SECTION=section-name,
**       STRING=character-string
```

The values for the CSCAN Function are:

project-name - name of the project containing the section to be scanned.

library-name - name of the library containing the section to be scanned.

section-name - name of the section to be scanned.

character-string - the specific character string to search for, enclosed in quotes if special characters occur within the string. See Section 3.4 for details on when quotes are required; maximum of 48 characters.

An example of a CSCAN Function is:

```
** PARAM PROJ=FHACD129,LIBR=NEWCODE,SECTION=JOB
** CSCAN STRING="ADUN"
```

3.4.8 DOCUMENT

The DOCUMENT Function is used to print documentation stored in a library in the form of program design language, structured source code, text, etc. Thus this function can be used with units which are in card-image format; that is, units in the SOURCE, PDL, LINK, JOB, TEST, MGMT, and TEXT sections. Output details are provided on Subfunction cards (HEADER, TEXT, EJECT, and SPACE) which follow the DOCUMENT card.

```
 **   DOCUMENT   PROJECT=project-name,
 **              LIBRARY=library-name,
 **              SECTION=section-name,
[**              LSPACE=line-spacing,]
[**              PAGE=    {NO       }  ]
[                         {page-nbr } ,]
[**              PLINES=lines-per-page,]
[**              START=start-line-nbr,]
[**              USPACE=unit-spacing,]
[**              LADJUST= {YES}  ]
[                         {NO }  ]
```

The values for the DOCUMENT parameters are:

project-name - name of project

library-name - name of library

section-name - name of section; must be SOURCE, PDL, LINK, JOB, MGMT, TEST, or TEXT

line-spacing - the number which represents the line spacing required for text; the number 1 represents single spacing of text; 2 represents double spacing, etc., the default value is 1 (single spacing).

page-nbr - the starting page number of the document; the default is 1; if the printing of page numbers is to suppressed, the word NO is to be used instead of a number.

lines-per-page - maximum number of lines which can be printed on a page (including headers and blank lines); the default is 50.

left-adjust - specified whether document text is to be left adjusted to left margin of page or to be placed in center of the 132 character line.

start-line-nbr - the starting line number for any printing on a page, whether Header lines or text; the default value is 1.

unit-spacing - the number which represents the line spacing between units. The number 1 represents single spacing (the next unit immediately follows the previous line), 2 represents double spacing, etc.; if cmitted, it is set equal to the value for LSPACE.

Optional subfunction cards are used to provide details for the DOCUMENT output requirements. The optional keywords for the DOCUMENT function (LSPACE, PAGE, PLINES, START, USPACE, LADJUST) may also be used on the three subfunction cards (HEADER, TEXT, and SPACE) but not on the EJECT card.

In addition to the five previously mentioned optional keywords, the values for the Subfunction keywords are:

```
**  HEADER     HPRINT=  { NO  }
                        { YES }
               HSPACE=header-spacing
(source cards for new Header may be placed here)
**  TEXT       UNIT=unit-name
(source cards for additional text may be placed here)
**  SOURCE     LINES=nbr-of-spaces
**  EJECT      (no keywords are associated with the
               EJECT Subfunction)
```

HPRINT - NO is used to stop the print of Header lines which had previously been input; YES is used to restart the printing of Header lines which had been suppressed by a prior HPRINT=NO keyword. When SOURCE cards for a new header are entered, the default value is YES.

header-spacing - The number which represents the line spacing between the last Header line and the first line of text on a page. The number 1 is single spacing (text immediately follows the header); 2 represents double spacing, etc.

header source cards - The provided cards are printed at the top of each new page on the DOCUMENT Function output. A maximum of ten cards may be used as input; single spacing is assumed on output, therefore a blank line may only be placed in the header by including a blank card.

unit-name - Name of the unit which is to be printed; the unit must be present in the section and must not be a Stub unit. If the unit desired is located in a different library and/or section, then a new DOCUMENT Function card must be used. Data stored in a unit will be printed after any source card text is printed.

text source cards - Any number of source cards may be placed in the run stream following a TEXT card; if the TEXT card contains the optional UNIT parameter, the named unit will be printed after the source card text.

nbr-of-spaces - Actual number of blank lines desired.

3.4.9 EXECUTE

The EXECUTE Function invokes the execution of a user's program. The program itself may be retrieved by the PSL system in one of the following forms:

a. A load module from the PROGRAM section

b. A series of one or more object modules, which are selected as a result of collector control cards in a unit of the LINK section and which will be processed by the Collector.

The user's job control cards for execution of his program are stored in a unit in the JOB section. Special PSL flags are placed in these cards to direct the system in the placement of additional file references.

```
  **   EXECUTE    PROJECT=project-name,
  **              LIBRARY=library-name,
  **              JOB=job-unit-name,
 {[**             LOAD=load-unit-name,]}  only one will be
 {[**             LINK=link-unit-name,]}  used
  [**             PROJ1=first-project-to-search,]
  [**             PROJ2=second-project-to-search,]
                       .
                       .
  [**             PROJ9=ninth-project-to-search,]
  [**             LIB1-first-library-to-search,]
  [**             LIB2=second-library-to-search,]
                       .
                       .
  [**             LIB9=ninth-library-to-search]
```

The values for the execute parameters are:

project-name - name of the first project to search for the JOB, LOAD, or LINK units; equivalent to value for PROJ1.

library-name - name of the first library, to search, as above; equivalent to LIB1.

job-unit-name - name of unit containing the user's JCL for execution of his program; see discussion of PSL flags below.

load-unit-name - name of unit containing the system-loadable program; only one unit-name will be used from the LOAD or LINK sections; LOAD unit created by LINK Function; default is load-unit-name equals job-unit-name.

link-unit-name - name of unit containing Collector control cards to load OBJECT modules for execution; the load module is not saved; see LINK Function for description of PSL OBJECT INCLUDE cards (which are used with collector control cards) and for discussion on use of LINK unit.

first-project-to-search - equivalent to value for PROJECT; if both are entered, the last one will be used.

second-project-to-search - see subsection 3.2.2 for a discussion of a multi-library search.

.

.

ninth-project-to-search

first-library-to-search - equivalent to value for LIBRARY; if both are entered, the last one will be used.

second-library-to-search - see subsection 3.2.2 for a discussion of a multi-library search.

.

.

ninth-library-to-search

The job control cards stored in a unit in the JOB section will be retrieved by the PSL system and used as the basic input stream for the execution. However, some information must be added to this input stream by the PSL system. In order to direct the placement of this information, special PSL flag-words are inserted in columns 73 through 80 (left-justified) on the user's control cards.

For the EXECUTE Function, the user inserts the word EXECUTE on the first card of his execution activity:

```
Col                                     Col
1                                       73
@XQT                                    EXECUTE
```

3.4.10 INDEX

The INDEX Function prints a status report for a section. The report contains information pertaining to the section as a whole, as well as a listing of the index for the section. An example of a Section Index Report is shown in Figure 3-02.

```
**  INDEX   PROJECT=project-name,
**          LIBRARY=library-name
**          SECTION=section-name
```

The values for the INDEX Function are:

project-name - name of project.

library-name - name of library.

section-name - name of section for which report is printed.

3.4.11 INITIAL

Under the PSL system, a user stores programs and data in discrete units, within sections of a library, under a project. A project must be initialized before libraries are built under it. The project identification number is used as the name of the project. The PSL Function INITIAL establishes the project as a PSL project and prepares an index for library-sections under the project.

```
     **  INITIAL    PROJECT=project-name,
    [**             FMS=fms-entry]
```

The values for the INITIAL Function are:

project-name - name for project to be initialized under the PSL system; maximum of twelve characters.

fms-entry - used to enter FMS parameters for random index file for project; default size is two tracks reserved and maximum.

3.4.12 JCL

The JCL Function enables the user to introduce JCL cards into the input stream of a subsequently spawned job. The Function is not required in the basic use of the PSL system, but provides the flexibility of using additional Exec 8 control cards in conjunction with such PSL Functions as COMPILE and LINK. See subsection 3.2.5 for a discussion of spawned jobs.

```
     **  JCL
```

No keywords are associated with the JCL Function.

The following example adds a new activity to the spawned job:

```
     **  PARAM      PROJ=UMC1234,LIBR=MYLIB
     **  LINK       LINK=TIPTOP
     **  INDEX      SECT=LOAD
     **  JCL
     @PRT,F    MYFILE.
     @XQT      MY.JOB
```

A @FIN card should not be placed in the cards added with the JCL Function. The PSL system will add a @FIN card at the end of the last activity in the spawned job.

3.4.13 LINK

The LINK Function retrieves one or more object module entries from the OBJECT section, collects the corresponding relocatable elements, and records the new absolute element in the LOAD section. The resulting absolute element is stored as a unit in the PROGRAM section. The name of the LOAD unit is the same as that of the LINK unit.

```
 **   LINK  PROJECT=project-name,
 **         LIBRARY=library-name,
 **         PASSWORD=load-section-password,
[**         LINK=link-unit-name,]
[**         OBJECT=object-unit-name,]
[**         PROJ1=first-project-to-search,]
[**         PROJ2=second-project-to-search,]
              .
              .
              .
[**         PROJ9=ninth-project-to-search,]]
[**         LIB1=first-library-to-search,]
[**         LIB2=second-library-to-search,]
              .
              .
[**         LIB9=ninth-library-to-search]
```

The values for the LINK parameters are:

project-name - name of first project to search for LINK unit when required and name of project to be used for same resulting LOAD unit; equivalent to value for PROJ1.

library-name - name of first library to search for LINK unit when required and name of library to be used for storing resulting LOAD unit; equivalent to value for LIB1.

load-section-password - password of LOAD section; not required if password not assigned to section.

link-unit-name - name of unit containing collector control cards.

object-unit-name - name of the unit recorded in the OBJECT section.

first-project-to-search - equivalent to value for PROJECT; if both are entered, the last one will be used.

second-project-to-search - see subsection 3.2.2 for a discussion of a multi-library search.

.

.

ninth-project-to-search

first-library-to-search - equivalent to value for LIBRARY; if both are entered, the last one will be used.

second-library-to-search - see subsection 3.2.2 for a discussion of a multi-library search.

.

.

ninth-library-to-search

The LINK unit must meet the following requirements:

The collector control cards are scanned for special PSL OBJECT INCLUDE cards, which are formatted as follows:

INCLUDE object-unit-name

The word INCLUDE may start in any column. The object-unit-name must terminate before column 73 and must be preceded by at least one space. The OBJECT module referenced by object-unit-name is located in the single or multi-libraries declared with the PROJECT, LIBRARY, etc., keywords. The PSL system changes each include card to an IN statement with the proper PROJECT file and relocatable element. The resulting LOAD unit will be given the same name as the LINK unit. If an OBJECT module is not found in the library, a dummy OBJECT module (OBJECT stub) is generated with the object-unit-name of the OBJECT INCLUDE card. If the module is called during execution, the OBJECT stub prints the message:

STUB (object-unit-name) CALLED

Also, if the first OBJECT INCLUDE card in the unit is a stub, a MAIN stub will be generated.

3.4.14 MDCOLLECT

The MDCOLLECT Function is used to collect management data in accordance with requirements of the Management Data Plan (provided for through use of the MDPLAN Function) and the management data format specifications (provided for thorough use of the MDFORMAT Function). Automatically maintained management data is derived from MDPLAN-specified source code modules while manually input data is assembled from MDPLAN-specified units established and updated through use of the MDUPDATE Function.

```
 **  MDCOLLECT  PROJECT=project-name,
 **             LIBRARY=library-name,
 **             PASSWORD=section-password,
 **             KEY=unit-key,
[**             ARCHIVE=   {YES}  ]
[                          {NO }  ,]
[**             RECYCLE=   {YES}  ]
[                          {NO }  ,]
[**             PGMR=programmer-name,]
[**             PROCEDURE=special-collection-procedure,]
```

The values for the MDCOLLECT parameters are:

project-name - name of project.

library-name - name of library.

section-password - password which was assigned when the management section was created; not required if a password was not assigned.

unit-key - key to be assigned (when MDCOLLECT Function is first used) or key which was assigned (when MDCOLLECT Function is subsequently used); not required if a key was not assigned when MDCOLLECT was first used.

ARCHIVE - if not specified, archiving of the previously collected data will depend upon parameters established in the relevant format level units; if YES is specified, archiving will occur as permitted for each format level; if NO is specified, archiving will not occur.

RECYCLE - if not specified, recycling of cyclic data accumulations will depend upon parameters established in the relevant format level units; if YES is specified, recycling will occur for all referenced format levels and archiving may occur (unless otherwise specified); should NO be specified, recycling will not occur nor will archiving (unless otherwise specified).

programmer-name - name to be placed in accounting record; maximum of twelve characters; default is userid for job.

special-collection - the name to be used to invoke a special JCL procedure to collect management data, instead of the standard procedure stored in the system. This special procedure must also have been previously stored in the system in order to be invoked; (see Installation Manual).

3.4.15 MDFORMAT

The MDFORMAT Function is used to provide the ability to add, change, or delete management data format units in the Management (MGMT) Section. A unit listing is automatically produced after MDFORMAT is completed.

```
    **   MDFORMAT    PROJECT=project-name,
    **               LIBRARY=library-name,
    **               PASSWORD=password,
                     KEY=key,  (          )
                               |SYSTEM    |
                               |SUBSYSTEM |
    **               LEVEL=    {MODULE    }  ,
                               |JOB       |
                               (UNIT      )
   [**               MOD=modification-number,]
   [                     (ADD    )     ]
   [**               OP= {CHANGE }     ]
   [                     {DELETE }  ,  ]
   [                     (MOVE   )     ]

   [**               CYCLE=cycle-period,]
   [**               OLDPROJ=old-project-name,]
   [**               OLDLIB=old-library-name,]
   [**               ARCHIVE= {YES}  ,]
   [                          {NO }   ]
   [**               PGMR=programmer-name]
```

The values for the MDFORMAT Function are:

project-name - name of management data project.

library-name - name of management data library.

section-password - password which was assigned to MGMT Section when it was created; not required if a password was not assigned.

unit-key - when OP=ADD, key to be assigned to the unit for use in subsequent updates. When OP=CHANGE, DELETE or MOVE, key assigned when unit was added; not required if no keyword was assigned when unit was added.

modification-number - Number to be checked against modification level in accounting record; a listing of the unit is printed but no update takes place if values do not match; maximum value of 999.

OP - The operation code, indicates whether a format unit is to be added, deleted, changed or moved. The default is CHANGE. If MOVE is specified, then OLDPROJ or OLDLIB keywords is required.

LEVEL - The format level for which management data items are defined.

cycle-period - A number (not to exceed two digits) indicating the minimum calendar-day period constituting a cumulative data cycle. Default value is zero, indicating that the cycle period is undefined.

old-project-name - Name of project from which the format is to be moved.

old-library-name - Name of library from which the format is to be moved.

ARCHIVE - Archiving of data items defined by the format unit is permitted if the value is YES; archiving is not permitted if the value is NO. The default (if LEVEL=SYSTEM) is YES, otherwise the default is NO.

programmer-name - Name to be placed in accounting record; maximum of twelve characters; default is userid for job.

The following examples show how to add a format unit under one project library and subsequently move it to another:

```
** PARAM      PROJ=SYSTEM,LIBR=PSL
** MDFORMAT   OP=ADD,LEVEL=SYSTEM
(source transactions follow)
```

```
**   PARAM       PROJ=UMC1234,LIBR=MYLIB,
**   MDFORMAT    OP=MOVE,LEVEL=SYSTEM,CYCLE=28,
**               OLDP=SYSTEM,OLDL=PSL
```

Note that the cycle period is updated subsequent to the move (as would the archive indication if it were specified).

The following example will move the level format specified on the old project and old library to the management data project and library:

```
**   PARAM       PROJ=UMC1234,LIBR=NEWCODE
**   MDFORMAT    OP=MOVE,
**               OLDPROJ=UMC0111,OLDLIB=OLDCODE
```

Format specifications are entered as input data in card columns 1-72. Figure 3-13 contains the definition of the format specifications.

Appendix J contains a SOURCE Function printout of the MDCR Format units delivered with the PSL system and stored in the system project PSL library MGMT section. These formats may be moved to a user-designated MGMT section and either subsequently modified or used "as is" to facilitate the initial utilization of the MDCR capability.

Card Column	Data Name	Data Description
1	Operation Code	I (Insert) M (Modify) D (Delete)
2-4	Item Number	a three digit numeric; zero is not permitted.
5	Data Source Code	b (manual input data) * (an item in the same format) S (an item in subsystem format) M (an item in module format) J (an item in job format) A (an item in the accounting item list) - refer to Tables 3-1 and 3-2. $ (an item in the special item list) - refer to Table 3-2.
6-8	Referenced Item (if column 5 not=blank)	a three digit numeric that matches an item number in the source format or list)
6-8	Repeated Item Code (if column 5=blank)	specified as "RPT" for manual input items only.

9	Manual Input Edit	5(no edit)
		A(alphabetic)
		N(numeric)

10-11	Maximum Length (if column 5=blank)	two digit numeric from 01 to 48; defaults to 48 unless column 9=N; numeric values will default to 08 which is the maximum length that a numeric value can be assigned.

Figure 3-13. Management Data Format Specifications

Card Column	Data Name	Data Description
9-11	Summary Function (if column 5 not=blank)	AVG (computes average input value) MAX (computes maximum input value) TTL (computes total of input values) - default value
9-11	Cyclic Function (if column 5=*)	CYC (computes the cumulative change in value of the Referenced Item specified in columns 6-8)
12-23	Item Name	mnemonic name of defined items; must be left justified and start with an alphabetic character. No embedded blanks are permitted.
25-72	Item Label	a descriptive label used in output reports to fully identify the reported item value.

Figure 3-13. Management Data Format Specifications (Continued)

Item Referenced	Unit Accounting Data Input
A001	Unit Type Name
A002	Unit Line Size
A003	Including Unit Name
A004	Version Number
A005	Modification Number
A006	Date Unit Originated
A007	Originator Name
A008	Date Last Update
A009	Time Last Update
A010	Unit Language
A011	Include Count
A012*	Total Lines in Unit
A013*	Structured Program Error Code
A014*	Lines Added
A015*	Lines Changed
A016*	Lines Deleted
A017*	Total Lines Input
A018*	Lines Input in Cycle
A019*	Lines Copied
A020*	Total Number of Updates
A021*	Updates in Cycle
A022*	User Work Area

* - Extended MGMT Data

Table 3-1. Unit Format References

Item Referenced	Applicable Format/Item Category: Data Input
	Module Format/Summary Input Items
A101	Lines in Unit
A102	Unit Lines Added
A103	Unit Lines Changed
A104	Unit Lines Deleted
A105	Unit Lines Input - Total
A106	Unit Lines Copied
A107	Unit Updates - Total
	Module Format/Numeric Items
A201	Compiles - Total
A202	Real Unit Count
A203	Stub Unit Count
A204	Structured Program Unit Error Count
	Module Format/Non-Numeric Items
A301	Top Unit Type
A302	Date Top Unit Originated
A303	Originator Name
A304	Top Unit Language
A305	Date Module Last Updated
	Non-Unit Formats/Special Items
$001	Subordinate Element Count
$002	Start Cycle Date
$003	Cycle Duration (days)
$004	Current Date
$005	End Cycle Indication (No=0, Yes=1)

Table 3-2. Module Accounting and Special Item References

When a User wishes to generate a report other than those produced by the PSL, he uses the MDPRINT Function to provide the printing capability.

```
** PARAM      PROJ=UMC1234,LIBR=MYLIB
** MDFORMAT   OP=MOVE,LEVEL=SYSTEM,CYCLE=28,
**            OLDP=SYSTEM,OLDL=PSL
```

Note that the cycle period is updated subsequent to the move (as would the archive indication if it were specified).

The following example will move the level format specified on the o ' project and old library to the management data project and library:

```
** PARAM      PROJ=UMC1234,LIBR=NEWCODE
** MDFORMAT   OP=MOVE,
**            OLDPROJ=UMC0111,OLDLIB=OLDCODE
```

Format specifications are entered as input data in card columns 1-72. Subsection 3.4.15 contains the definition of the format specifications.

Appendix J contains a source listing of the MDCR-FORMAT units delivered with the PSL system and stored in the System project PSL library MGMT Section. These formats may be moved to a user-designated MGMT Section and subsequently modified or immediately used to facilitate initial utilization of the MDCR capability.

3.4.16 MDPLAN

The MDPLAN Function is used to modify the contents of the Management Data Plan (MDPLAN) unit which is automtically established (without data content) when a Management Section is created. A unit listing is automatically provided for the indicated plan unit. Modification details are provided on Subfunction cards following the MDPLAN card exactly as prescribed for the CHANGE Function.

```
 **  MDPLAN     PROJECT=project-name,
 **             LIBRARY=library-name,
 **             PASSWORD=section-password,
 **             KEY=plan-unit-key,
[**             MOD=modification-level,]
[**             PGMR=programmer-name]
```

The values for the MDPLAN parameters are:

project-name - Name of project.

library-name - Name of library.

section-password - Password which was assigned when the management section was created; not required if password was not assigned.

plan-unit-key - Key assigned when management section was created; not required if a key was not assigned.

modification-level - Number to be checked against the modification level in the unit accounting record; no update takes place if values do not match; maximum value of 999.

programmer-name - Name to be placed in accounting record; maximum of twelve characters; default is userid for job.

The Subfunctions which are used to alter the contents of an MDPLAN unit are described under the CHANGE Function. The following example shows an appropriate procedure for providing the initial input to that unit:

```
**  PARAM       PROJ=UMC1234,LIBR=NEWCODE
**  CREATE           SECTION=MGMT,KEY=SKELETON
**  MDPLAN           KEY-SKELETON
**  I A=0
```

(source cards are initially inserted at beginning of unit; card columns 1-72 are used for data input)

Sample input:

```
    SYSTEM=BIGTOP
    SUBSYS=SUB1,PROJ=UMC001,LIBR=TEST
      MODULE=MOD1A
      MODULE=MOD2A,LIB2=PROVEN
    SUBSYS=SUB2,PROJ2=UMC002,LIB2=INTEG
      MODULE=MOD2A,MODULE=MOD2B
```

Project/library keyword value specifications are cumulative and may be changed at any point in the plan input sequence. Project and library keyword values may be nulled by omitting the keyword value specification (e.g., PROJ$_n$=,LIB$_n$=). Keyword/value specifications for a given input line can have no imbedded blanks; however, blank lines may be inserted as needed.

Keyword specification may begin in any column as long as the given keyword and value specification is wholly contained on a single card. Project and library specification applying to a given module may be given prior to the occurrence of the module element or given "in continuation" of the module element specification by using commas to separate successive project and library keyword specification that apply to that module. Verification of the plan syntax is performed when the MDCOLLECT Function is used.

3.4.17 MDPRINT

The MDPRINT Function is used to print management data reports. Keywords for the MDPRINT Function consists of both general keywords, which are processed by the MDPRINT processor, and specific report keywords, which are passed to a spawned report generator. See subsection 3.2.5 for a discussion of spawned jobs.

```
 **  MDPRINT    PROJECT=project-name,
 **             LIBRARY=library-name,
 **             SECTION=section-name,
 **             REPORT=report-code,
[**             PROJ1=first-project-to-search,]
[**             PROJ2=second-project-to-search,]
                  .
                  .
[**             PROJ9=ninth-project-to-search,]
[**             LIB1=first-library-to-search,]
[**             LIB2=second-library-to-search,]
                  .
                  .
[**             LIB9=ninth-library-to-search]
```

The values for the general keywords for the MDPRINT Function are:

project-name - Name of the first project to search; equivalent to value for PROJ1.

library-name - Name of first library to search; equivalent to LIB1.

section-name - Name of section, if appropriate; in the case of PS report should be PDL or SOURCE.

report-code - Two-letter code to select report.

first-project-to-search - Equivalent to value for PROJECT; if both are entered, the last one will be used.

second-project-to-search - See subsection 3.2.2 for a discussion of a multi-library search.

.

.

ninth-project-to-search

first-library-to-search - Equivalent to value for LIBRARY; if both are entered, the last one will be used.

second-library-to-search - See subsection 3.2.2 for a discussion of a multi-library search.

.

.

.

ninth-library-to-search

Program Structure Report

Keywords for Program Structure (PS) report;

```
     **  UNIT=beginning-unit-name,
    [**  CALL=  {NO }  ,]
    [           {YES}    ]
    [**  NEST=  {50 }    ]
    [           {nbr}    ]
```

The values for the PS keywords are:

beginning-unit-name - Name of unit with which to begin the hierarchical search; not required to be a top-unit.

CALL=YES - If option is selected, a PS report will be generated for each unit referenced by a CALL statement in the code of the units scanned.

NEST=nbr - If option is selected, the hierarchical search will not scan units for which the INCLUDE statements are nested at levels deeper than this value; the default of 50 is the PSL limit for nested INCLUDE statements.

The PS report begins with the unit which is named on the Function card and develops a hierarchically nested and indented list of all units which are referenced by an INCLUDE statement, either in the originally-requested unit or in units which are themselves INCLUDEd in the hierarchical program structure. Units which are referenced by a CALL statement are also printed with the appropriate indentation in the list, but they are not automatically searched for lower levels of INCLUDE or CALL statements. Statistical and organization information is printed for each unit. An alphabetically-arranged list of all referenced units follows the hierarchical list. An example of a Program Structure Report is shown in Figure 3-04.

```
Example:
**  MDPRINT    PROJECT=UMC1234,LIBRARY=MYLIB,
**             UNIT=TIPTOP,REPORT=PS
```

Management Data Report

The specific keywords for the Management Data (MD) report are:

```
[ ** SYSTEM=    {YES} ,]
[               {NO }  ]
[ ** SUBSYSTEM= {YES}  ,]
[               {NO }   ]
[ ** JOB=       {YES}  ,]
[               {NO }   ]
[ ** MODULE=    {YES}  ,]
[               {NO }   ]
[ ** UNIT=      {NO }  ,]
[               {YES}   ]

[ ** ELEMENT=beginning-element-name,]

[ ** HISTORY= {NO                                  } ]
[             {ALL                                 } ]
[             {(SERIES,start-series, end-series)   } ]
[             {(DATES,start-date, end-date)        } ]
```

The values for the MD keywords are:

SYSTEM=NO − If option is selected, system level reports will be suppressed.

SUBSYS=NO - If option is selected, subsystem level reports will be suppressed.

MODULE=NO - If option is selected, module level reports will be suppressed.

JOB=NO - If option is selected, job level reports will be suppressed.

UNIT=YES - If option is selected, unit level reports will be enabled.

beginning-element name - Name of report level element with which to begin the hierarchical output; a report will be produced for the specified element (if present) and for subordinate level elements in the hierarchy, provided that reports are not suppressed at the subordinate level(s). The ouput of all other report elements will be suppressed.

HISTORY=ALL - If this option is selected, all archived data collections in the MGMT Section will be processed for output.

HISTORY=(SERIES...) - If this option is selected, the archived data collections included in the designated series will be processed for output.

HISTORY=(DATES...) - If this option is selected, the archived data collections whose data was collected within the calendar period defined by the specified dates will be processed for output.

start-series - The serial number of the first archived data collection to be processed; minimum value=0.

end-series - The serial number of the last archived data collection to be processed; maximum value=999.

start-date - The earliest date of data collection that will be processed for output; date format is MMDDYY.

end-date - The latest date of data collection that will be processed for output; date format is MMDDYY.

3.4.18 MDUPDATE

After required format units are added to the MANAGEMENT section, the MDUPDATE Function is used to originally introduce data into a management data unit, modify the contents of a management data unit, or delete a management data unit. Modification details are provided by the operation-code and the Subfunction cards (DELETE, MODIFY, and INSERT) which follow the MDUPDATE card. New management data follows either the MDUPDATE card for the add operation or the INSERT and MODIFY Subfunction cards for the change operation. Identification of management data items to be deleted follow the DELETE Subfunction card.

```
 **  MDUPDATE   PROJECT=project-name,
 **             LIBRARY=library-name,
 **             PASSWORD=section-password,
 **             UNIT=management-data-unit,
[**             LEVEL=unit-level,]
[**             MOD=modification-level,]
 **             KEY=unit-key
 **             OP=operation-code
```

The value for the MDUPDATE parameters are:

project-name - Name of project.

library-name - Name of library.

section-password - Password which was assigned when the MGMT section was created; not required if a password was not assigned.

management-data-unit - Name of management data unit to be processed.

unit-level - Level of management data unit; must be SYSTEM, SUBSYSTEM, MODULE, or JOB; required only if operation-code is ADD.

modification-level - Number to be checked against modification level in unit accounting record; a listing of the unit is generated but no update takes place if values do not match; maximum value 999.

unit-key - Key which is assigned to unit when it is added to MGMT section; not required if a key is not assigned.

operation-code - The operation to be performed by the MDUPDATE Function; must be ADD, CHANGE or DELETE.

The Subfunction cards are used only when the change operation is specified on the MDUPDATE card.

```
** INSERT
   (new management data)
** MODIFY
   (management data to be modified)
** DELETE
   (management data identifiers to be deleted)
```

The Subfunctions need not be in a particular order. The management data will be edited, verified, and sorted before the management input unit is processed by the MDUPDATE Function. All Subfunctions may be abbreviated to four characters or to one character.

The MDUPDATE Function updates a management data unit by processing data items supplied in the input data and by creating a new copy of the unit. After the MDUPDATE processing has terminated, a listing of the unit is automatically generated showing the actual contents of the unit in the library. The following examples show how a user may add, change, or delete the contents of a management data unit:

```
** PARAM      PROJ=UMC1234,LIBR-NEWCODE
** MDUPDATE   UNIT=TIPTOP-MANAGEMENT-DATA[1],OP=ADD,
**            LEVEL=SYSTEM,KEY=HIDDEN
   (new management data)
** MDUPDATE   UNIT=TIPTOP-MANAGEMENT-DATA,OP=CHANGE,
**            KEY=HIDDEN
** M
   (management data to be modified)
** I
   (new management data to be inserted)
** D
  (management data to be deleted
** MDUPDATE   UNIT=TIPTOP-MANAGEMENT-DATA,OP=DELETE
**            KEY=HIDDEN
```

[1] Unit must not exist prior to add operation.

Input data may utilize card columns 1-72 to provide manual input data. The forms in which theinput data may be submitted are as follows (no imbedded blank permitted):

```
**  I or ** M (for a non-repeated item)
    item-ident=item-value
**  M (for a repeated item)
    item-ident,repeat-nbr=item-value
**  D (for a non-repeated item)
    item-ident
**  D (for a repeated item)
    item-ident,repeat-nbr
```

in which the following descriptions apply:

item-ident - The format item number <u>or</u> item-name.

item-value - The value with which the identified item is to be updated (it should conform to the format Edit specification).

sequence-nbr - A two digit number assigned to repeated occurrences of a given item-ident by MDUPDATE when a format specified RPT item is multiply submitted following an Insert subfunction.

An RPT specified item (see MDFORMAT Function) is one for which multiple values may be submitted. Each inserted item value is identified when a sequence number one higher than the previously inserted value for that same item. In order to modify or delete a specific occurrence of the repeated item, the corresponding sequence number must be obtained from the unit listing whose format is illustrated in Figure 3-13. As noted, the sequence number (in columns 10-11) immediately follows the item number (in columns 7-9).

3.4.19 MDXCHECK

After a management data unit has been added to the MGMT section, the MDXCHECK Function is used to add, change or delete exception checking specifications within the existing management input unit. Modification details are provided by the Subfunction cards (DELETE, MODIFY, and INSERT) which follow the MDXCHECK card. New and revised exception checking specifications follow the INSERT and MODIFY Subfunction cards, respectively. Identification of exception checking specifications to be delted follow the DELETE Subfunction card.

```
 **  MDXCHECK   PROJECT=project-name,
 **             LIBRARY=library-name,
 **             PASSWORD=section-password,
 **             UNIT=management-data-unit,
[**             MOD=modification-level,]
[**             KEY=unit-key]
```

The values for the MDXCHECK parameters are:

project-name - Name of project.

library-name - Name of library where MGMT section exist.

section-password - Password which was assigned when the MGMT section was created; not required if a password was not assigned.

management-data-unit - Name of management data unit to be processed; unit must have been added to the MGMT section prior to MDXCHECK processing.

modification-level - Number to be checked against modification level in unit accounting record; a listing of the unit is generated but no update takes place if values do not match; maximum value 999.

unit-key - Key which was assigned to management data unit when it was added to MGMT section; not required if a key was not assigned.

The Subfunction cards are required to follow the MDXCHECK card.

```
     **   INSERT
          (new exception checking data)
     **   MODIFY
          (exception checking data to be modified)
     **   DELETE
          (exception checking data identifiers to be deleted)
```

The Subfunctions need not be in a particular order. The exception checking data will be edited, verified and sorted before the management data unit is processed by the MDXCHECK Function. All Subfunctions may be abbreviated to four characters or to one character.

The MDXCHECK Function updates an existing management data unit by processing exception checked data items against format-defined data items and creating a new copy of the management input unit.

After the MDXCHECK processing has terminated, a listing of the management input unit is automatically generated showing the actual contents of the unit in the library. The following examples show how a user may add, change or delete exception checking specifications in a management input unit:

```
     **   PARAM       PROJ=UMC1234,LIBR=NEWCODE
     **   MDXCHECK    UNIT=TIPTOP-MGMT-DATA[1],KEY=FIRST
     **   INSERT
          (new exception checking data)
     **   M
          (exception checking data to be modified)
     **   D
          (exception checking data identifiers to be deleted)
```

Input data cards may utilize columns 1-80 to provide exception check specifications. The form in which the input data must be submitted for added and modified exception check specifications is as follows:

$$\underline{\text{item 1}} \begin{Bmatrix} > \\ < \end{Bmatrix} \underline{\text{item 2}}\text{:}\quad \text{V } \underline{\text{percent}},\ \underline{\text{date}},\ [\underline{\text{remark}}]$$

where non-underlines delimiters are required as represented and underlined items are to be provided in accordance with the descriptions below.

[1] Unit must already exist prior to MDXCHECK Function processing.

The form in which input data must be submitted for a deleted exceptions check specification is as follows:

$$\underline{\text{item 1}} \begin{Bmatrix} < \\ > \\ , \end{Bmatrix} \underline{\text{item 2}}$$

where item 1 and item 2 must be given to identify an existing exception check specification.

3.4.20 MOVE

The MOVE Function moves a unit, from one project to another, from one library to another, or within a library.

```
     ** MOVE    PROJECT=receiving-project-name,
     **         LIBRARY=receiving-library-name,
     **         SECTION=section-name,
     **         PASSWORD=receiving-section-password,
     **         UNIT= {receiving-unit-name}  ,
                      {ALL                }
     **         KEY=receiving-unit-key,
   [ **         OLDPROJ=old-project-name,]
   [ **         OLDLIB=old-library-name,]
   [ **         OLDUNIT=old-unit-name,]
   [ **         REPLACE=  {NO }  ,  ]
   [                      {YES}     ]
   [ **         FMS=fms-entry,]
   [ **         PGMR=programmer-name]
```

The values for the MOVE Function are:

receiving-project-name - name of project to which unit is moved.

receiving-library-name - name of library to which unit is moved.

section-name - name of both receiving section name and old section name; must be JOB, LINK, PDL, SOURCE, TEST or TEXT.

receiving-section-password - password of receiving section; not required if one was not assigned.

receiving-unit-name - name of unit to which unit is moved; if UNIT=ALL, the PSL system will attempt to move all units in the section from old-library-name to receiving-library-name; see below for rules affecting a MOVE.

r eceiving-unit-key - key which was assigned to receiving-unit-name when it was added to the section; ignored if UNIT=ALL; applicable only if REPLACE=YES.

old-project-name - name of project from which unit is to be moved.

old-library-name - name of library from which unit is to be moved; required if UNIT=ALL.

old-unit-name - name of unit from which unit is to be moved; required if neither old-project-name nor old-library-name is used.

REPLACE=YES - if PSL system finds that a real unit already exists in receiving section, unit will not be moved unless YES option has been declared; NO is default; stubs will be replaced under either option.

fms-entry - used to enter FMS parameters for unit file, if unit is INDEPENDENT; default size for INDEPENDENT unit file is 1 reserved track, 2 maximum tracks.

programmer-name - name to be placed in accounting record of receiving unit(s); default is userid for job.

The simplest kind of MOVE involves a single unit being moved to a unit which does not exist. In this case, the unit-type and the cumulative accounting information is moved, as appropriate, with the unit.

If a single unit is moved to a unit which already exists as a real unit, the MOVE will not take place unless:

a. REPLACE=YES was declared.

b. The receiving-unit-key matches the key found in the receiving unit, if one was assigned.

c. The unit-type of the receiving unit is consistent with the unit-type of the old-unit (both are top-units, or both are included-units).

If the MOVE takes place, the resulting unit will retain the unit-type of the receiving-unit. The accounting information of the receiving-unit will be updated appropriately and INCLUDEs will be resolved within the receiving section.

When UNIT=ALL, an old-library-name and a new-library-name must be declared. The PSL system will attempt to MOVE each unit in the old library/section to the receiving library/ section. The rules described for a single-unit MOVE will be applied for each unit MOVE. Since key checking is not possible in this case, a unit will not be moved if the receiving unit was assigned a key.

A MOVE Function does not alter the status of the old location. Disposition of the old copy is at the discretion of the user.

For a move involving an independent unit, if the space assignment for the receiving unit is insufficient to hold all source data from the old-unit, the PSL will abort. In this case, the user should use the PURGE Function to purge the receiving unit from the section and rerun the job using the FMS keyword on the MOVE Function to assign a larger size to the independent unit file.

3.4.21 PARAM

This Function is used to enter high-level parameters which apply to a series of subsequent PSL Functions. Once the values are established, they remain in effect. All of the values, except security classification code, may be replaced by using the keyword again, on another Function card for which it is valid.

```
** PARAM
**        [PROJECT=project-name,]
**        [LIBRARY=library-name,]
**        [SECTION=section-name,]
**        [PASSWORD=section-password,]
**        [PGMR=programmer-name]
```

Each of these keywords will provide subsequent Functions with values. The user should refer to the descriptions of those Functions to understand how the value will be used. All keywords are optional with the PARAM Function.

3.4.22 PERFORM Function

The PERFORM Function invokes a designated procedure contained in a user's project library. The job control cards constituting this procedure specify the programs and files required for a particular user operation. The PERFORM Function enables a non-PSL program operation to interface with PSL data files by using flagword data (paced in columns 73 through 80 of a job control card) to direct the addition of a PSL data file reference. The subject data for that reference is taken from user-provided keyword input values and the subsequently modified job control procedure is written to the spawned job file.

```
     ** PERFORM       PROJECT=project-name,
     **               LIBRARY = library-name,
     **               PASSWORD = section-password
     **               PROCEDURE = user-procedure-name,
    [**               UNIT = unit-name,]
    [**               KEY = unit-key,]
    [**               FMS = fms-entry,]
    [**               FILE1 = first-user-file,
    [**               FILE2 = second-user-file,
                          .
                          .
                          .
                          .
                          .
    [**               FILE9 = ninth-user-file
```

The values for the PERFORM parameters are:

project-name - name of project.

library-name - name of library.

section-password - password of section(s) to be written by user procedure; not required if a password was not assigned to the section(s) when created.

user-procedure-name - name of unit in JOB section of designated project and library which contains all or part of the JCL required for a user-program operation.

unit-name - name of unit in PSL section(s) referenced by flagwords in the designated user-procedure.

unit-key - key to be assigned (when unit or file is first written) or key which was assigned (when unit is subsequently written); not required if a key was not assigned when unit was first written.

fms-entry - used to enter the FMS parameters for independent units/files which are referenced by flagwords in the designated user-procedure.

first-user-file - name of an independent file that is created and added to the directory maintained in the USER section of the designate project and library for access by user programs only.

second-user-file - names of additional files as may be required by the designated user procedure.
.
.
.
.
ninth-user-file

Flagword Card Format

User-procedure flagwords are left-justified to column 73 of a job control card. The prescribed format of the flagged job control card is as follows:

```
Col                         Col
1                           73
@ASG                        flagword [, access-code]
```

The following is a list of valid flagwords, subdivided into three data-reference categories.

PSL Section	User File	Card Input
JOB	FILE1	CARDS
LINK	FILE2	
LOAD	FILE3	
MGMT	FILE4	
OBJECT	FILE5	
PDL	FILE6	
SOURCE	FILE7	
TEST	FILE8	
TEXT	FILE9	
USER		

All flagwords <u>except</u> CARDS must be accompanied by one of the following access codes:

R: read permission requested

W: write permission requested

/: read and write permission requested

Flagwords in the PSL-section category are paired with the UNIT keyword input value to reference a unit in the noted PSL section, while in the user-file category, flagwords are corresponded with the matching keyword input to reference non-PSL data files that are accessed by user programs only. The PERFORM Function dynamically creates the referenced files (as needed) through interface with the Sperry Univac Exec 8 System provided that the required PSL section is created using STANDARD=NO as a section option.

<u>Replacement Card Format</u>

The job procedure card on which the flagword "CARDS" appears will be replaced by the following card:

Col 1	Col 73
@DATA,I	CARDS

User-provided card input data that follows the PERFORM Function keyword input specification will be inserted after the above @DATA card and succeeded by a @END card.

The job procedure card on which a valid PSL-section or user-file flagword appears will be replaced by a card in the following format:

```
Col                               Col
1                                 73

@ASG,A      file reference.       flagword specification
```

User File Directory

Files which are named through user-file keyword input values and designated for operations through user-file flagwords are accounted for in a section directory as established in the USER section of the keyword-referenced project and library. The CREATE Function must be utilized to establish the referenced USER section prior to the execution of the given PERFORM Function. Only that space required to maintain accounting data for the files need be requested when creating the USER section. The space required to create user files is obtained through the PERFORM Function, when needed, as directed by the accompanying FMS keyword input values or determined by the FMS default parameters assigned by PSL.

Although user-file contents are not directly accessed by PSL operations, it is necessary for the PERFORM Function to initiate user files, the TERMINATE and PURGE Functions to delete user files, and the INDEX Function to list user files. Neither the BACKUP option of the TERMINATE Function nor the BACKUP Function operate to save user-file data in the referenced USER section. Standard Univac 1100 utilities are available, however, that can be used to save the contents of user files should such backup be required.

3.4.23 PRECOMPILE Function

The PRECOMPILE Function retrieves units from the SOURCE section and invokes the appropriate precompiler. The unit name is the name of the top-most source unit of the module which is to be compiled. The processing which is invoked by the PRECOMPILE Function depends upon the language of the unit. Under the current PSL system, procedures exist to process the SCOBOL (Structured COBOL), SPFORT (Structured FORTRAN), SJOVIAL (Structured JOVIAL) and unstructured ASMG, COBOLG, FORTRANG, and JOVIALG languages. Procedures to precompile additional languages may be added to the system. Procedures are automatically invoked by the language-name, or may be specifically invoked with the PROCEDURE keyword.

```
     ** PRECOMPILE  PROJECT=project-name,
     **             LIBRARY=library-name,
     **             UNIT=unit-name,
    [**             PROCEDURE=special-precompile-procedure,]
    [**             PROJ1=first-project-to-search,]
    [**             PROJ2=second-project-to-search,]
                       .
                       .
                       .
    [**             PROJ9=ninth-library-to-search,]
    [**             LIB1=first-library-to-search,]
    [**             LIB2=second-library-to-search,]
                       .
                       .
                       .
    [**             LIB9=ninth-library-to-search]
```

The values for the PRECOMPILE parameters are:

project-name - name of first project to search for SOURCE units; equivalent to value for PROJ1.

library-name - name of first library to search for source units; equivalent to value of LIB1.

unit-name - name of top-most SOURCE unit to be compiled; unit must be MAIN, CALLED, or INDEPENDENT unit-type; maximum length of name is 6 characters.

special-precompile procedure - the name to be used to invoke a special JCL procedure to precompile the module, instead of the name of the language associated with the SOURCE code; this procedure must have been previously stored in the system; (see Installation Manual).

first-project-to-search - equivalent to value for PROJECT; if both are entered, the last one will be used.

second-project-to-search - see subsection 3.2.2 for a discussion of a multi-library search.

.

.

ninth-project-to-search

first-library-to-search - equivalent to value for LIBRARY; if both are entered, the last one will be used.

second-library-to-search - see subsection 3.2.2 for a discussion of a multi-library search.

.

.

.

ninth-library-to-search

The following example will invoke a precompile of a module using a multi-library search:

```
     **  PARAM         PROJ=FHACD129,LIBR=NEWCODE
     **  PRECOMPILE    UNIT=TIPTOP
     **                PROJ2=FHACD147,LIB2=PROVEN
```

Since TIPTOP is written in Structured COBOL (SCCBOL), the SCOBOL precompiler will process the units, starting with TIPTOP and working down through all INCLUDE statements. As each INCLUDE statement is picked up, the libraries are searched for the named unit. NEWCODE is searched first and then PROVEN. As each INCLUDEd unit is found, the code is read and added to the total stream of code which will be processed. Nested INCLUDEs are resolved in place so that the final module is in proper logical order. Comments are placed in the listing to show the project and library where each unit was found. If a unit is a SOURCE stub, a DISPLAY statement is inserted in its place. Original unit-line-numbers are placed in columns 1 through 6 of the precompiler output, for programmer reference.

The procedures which are provided with the PSL system are listed in Appendix D. The job cards in these procedures which occur before the compile activity (noted by the flagword COMPILE starting in column 73, when present) are output to the spawn job file to initiate the specified precompile. See subsection 3.2.4 for a discussion about spawned jobs.

The precompiler output will be stored in the temporary file denoted in the procedures as follows:

```
     @ASG,T     PO,F///100
```

Succeeding JCL can be added to the spawn job file via the JCL or PERFORM Functions which may utilize the temporary file as input to a succeeding activity. The precompile activity may insert an @USE PO.,PERMANENT-FILENAME and an @ASG,A PO. to override the @ASG,T card which then directs the precompiler output to a designated permanent file. See subsection 3.4.22 for an example of how the PERFORM Function may be used to direct precompiled output to specified PSL storage.

3.4.24 PURGE Function

An individual unit is removed from a library with the PURGE Function. A user should be aware that units are also removed from a library when a higher aggregation, such as a section or library, is removed with the TERMINATE Function.

```
** PURGE     PROJECT=project-name,
**           LIBRARY=library-name,
**           SECTION=section-name,
**           UNIT=unit-name, one only, required
**           FILE=file-name,
**           KEY=unit-key
```

The values for the PURGE Function are:

project-name - name of project.

library-name - name of library.

section-name - name of section.

section-password - password which was assigned when the section was created; not required if a password was not assigned.

unit-name - name of unit to be purged from the section index.

file-name - name of file to be purged from the section directory.

unit-key - key which was assigned to unit or file when it was added to section; not required if a key was not assigned.

Example:

```
** PARAM     PROJ=UMC1234,LIBR=MYLIB,SECTION=JOB
** PURGE     UNIT=TIPTOP,KEY=MYSTERY
```

When INDEPENDENT files are released, the FMS directive PURGE is used to overwrite the file space.

A unit whose type is STUB may not be removed from a section with the PURCE Function. When the last INCLUDE statement referencing the STUB unit is deleted from units in the section, the STUB unit will be automatically purged from the section.

3.4.25 REPLACE Function

After data has been added to a unit, the REPLACE Function is used to completely replace all of the data lines in the unit. Accounting information is not re-initialized, but is updated appropriately.

```
    ** REPLACE     PROJECT=project-name,
    **             LIBRARY=library-name,
    **             SECTION=section-name,
    **             PASSWORD=section-password,
    **             UNIT=unit-name,
   [**             KEY=unit-key,]
   [**             PGMR=programmer-name ]
```

The values for the REPLACE parameters are:

project-name - name of project.

library-name - name of library.

section-name - name of section; must be SOURCE, PDL, JOB, LINK, TEST or TEXT.

section-password - password which was assigned when the section was created; not required if a password was not assigned.

unit-name - name of unit to be replaced.

unit-key - key which was aesigned to unit when it was added to section; not required if a key was not assigned.

The REPLACE Function can be used to replace a real unit or a stub. However, if the unit does not exist in the section in any form, an ADD Function must be used.

If the space assigned to the file for an INDEPENDENT unit is insufficient to contain all replacement source data, the PSL program will abort. In that event, the user should use the PURGE Function to purge the old unit, and the ADD Function to put the new source data into the library with a larger space assignment. (See the description of ADD, Section 3.4.1).

3.4.26 RESTORE Function

The RESTORE Function is used to write the contents of the BACKUP tape into the library. The Function may be used to restore the complete contents of the BACKUP tape, or it may selectively recover a portion of the total data tape at a lower level.

```
** RESTORE     PROJECT=project-name,
               LIBRARY= {library-name} ,
                        {ALL         }
**             SECTION= {section-name} ,
                        {ALL         }
**             UNIT= {unit-name  ,}
                     {ALL         }
**             PASSWORD=section-password,
**             KEY=unit-key,
[**            FMS=fms-entry]
```

Plus a tape card with a file name of RT, as follows:

```
@ASG,A    RT,,reelnumber
```

This is a Exec 8 control card and follows the appropriate format. The tape card must be placed after the @END PSL card at the end of the PSL Function cards.

The values for the RESTORE parameters are:

project-name - name of the project; must be the same as the project identification on the @RUN control card for the job.

library-name - name of the library; ALL indicates that all libraries in the project are to be restored.

section-name - name of the section to be restored; ALL indicates that all sections in the library are to be restored; presence of this parameter is considered an error, if LIBRARY=ALL.

unit-name - name of unit to be restored; ALL indicates that entire section is to be restored; presence of this parameter is considered an error if LIBRARY=ALL or SECTION=ALL.

section-password - required for restore if single unit is password was assigned when section was created; not required if entire section is being restored.

unit-key - required for restore if single unit exists in section and key was assigned to unit in section; ignored if entire section is being restored.

fms-entry - used only when restoring units for which FMS data could not be saved on the backup tape by the BACKUP Function. This occurs most frequently when backing up a section or unit which was created with a project identification other than the one which is available on the backup tape, it will be used and user supplied values will be ignored. If FMS data is not available on the backup tape and there are no user supplied values, the PSL default values listed in Figure 3-10 will be used.

If all the units in a section are to be restored, the section must exist and must be empty, or the RESTORE will not be processed for that section. When a unit is restored, either singly or as part of a section, all data items for the unit, including statistical data and unit-key, are replaced by the data on the RESTORE tape.

The following example will invoke the PSL system and RESTORE the SOURCE section of the NEWCODE library:

```
Col
1
@USE          PSL.,DMA*PSL.
@ADD          PSL.RUN
** RESTORE    PROJ=FHACD129,LIBR=NEWCODE,
**            SECT=SOURCE,UNIT=ALL
@END          PSL
@ASG,A        RT,U9V,ZØ162Ø
@XQT          PSL.BCTL
```

3.4.27 SOURCE Function

The SOURCE Function can be used to print a listing of any unit which is in card-image format. This includes the unit in the SOURCE, PDL, LINK, JOB, TEST, MGMT, and TEXT sections. If the unit is from the SOURCE or the PDL section and the unit is written in a supported structured language (Structured COBOL, JOVIAL and FORTRAN), the structured code is automatically indented for the proper alignment of the structures on the listing. This listing is always provided when a unit is added to the library or modified. The contents of the unit may, on option, be written to tape or punched on cards. In the latter cases, the header information is omitted and the code is reproduced as it exists in the unit, without automatic indentation. An example of a listing of Structured COBOL (SCOBOL) is shown in Figure 3-03. An example of a listing of Structured FORTRAN (SPFORT) is shown in Figure 3-14.

```
 ** SOURCE      PROJECT=project-name,
 **             LIBRARY=library-name,
 **             SECTION=section-name,
 **             UNIT= {unit-name} ,
                      {ALL      }
[**             MEDIA= {LIST} ,  ]
[                      {TAPE}    ]
[                      {CARD}    ]
[**             NBRLINES= {50                } ,]
[                         {nbr-lines-per-page}  ]
[**             INDENT= {YES}  ]
[                       {NO }  ]
```

Plus a tape card with a file name of UT, if MEDIA=TAPE:

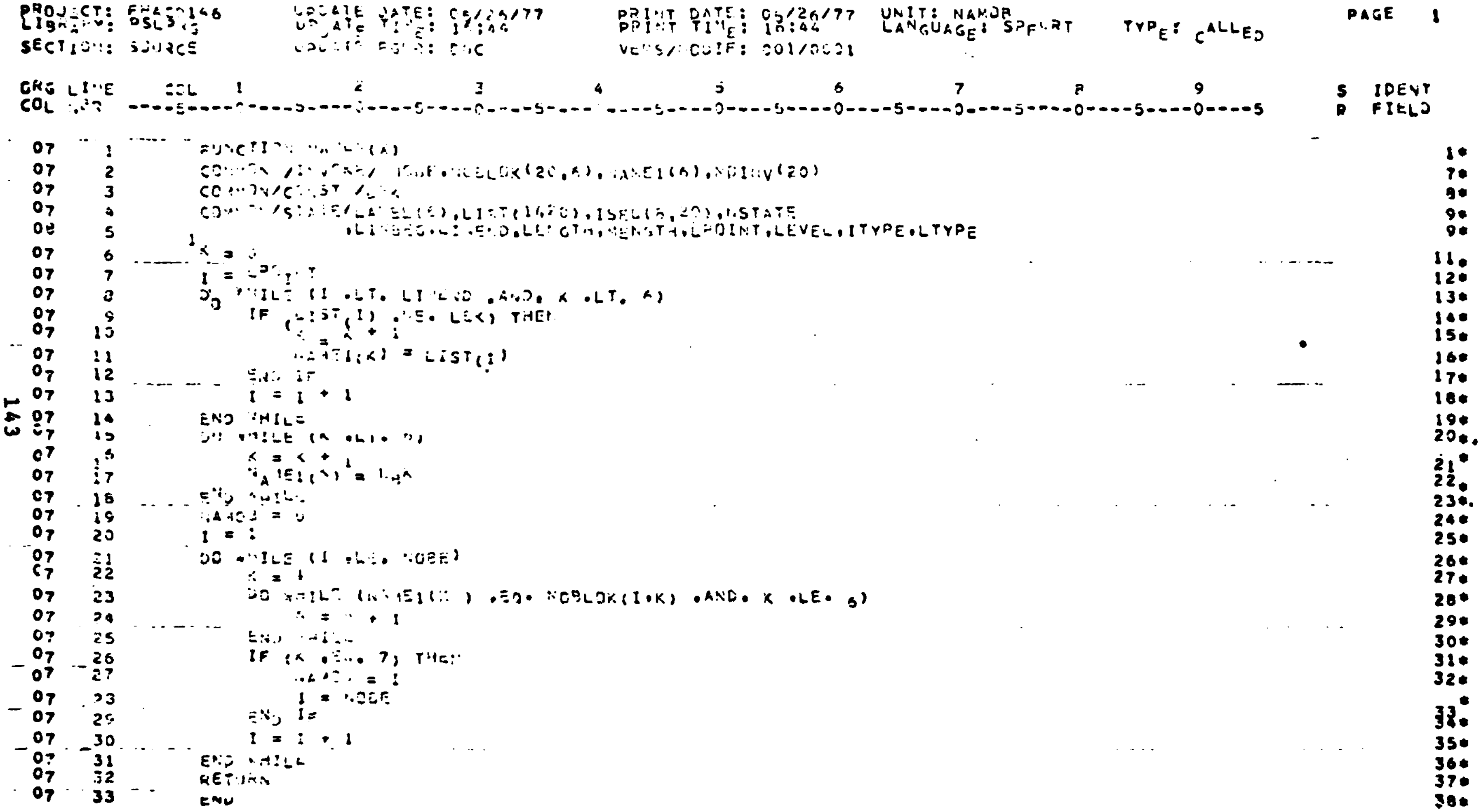
```
Col
1
@ASG,C     UT,U9V,Z01425
```

This is a Exec 8 control card and follows the appropriate format. The tape card must be placed after the @END PSL card at the end of the PSL Function cards. See BACKUP for example.

The values for the SOURCE parameters are:

project-name   -   name of project.

library-name   -   name of library.



```
PROJECT: FHAS0146        UPDATE DATE: 05/26/77      PRINT DATE: 05/26/77  UNIT: NAMOR                                  PAGE    1
LIBRARY: PSL3            UPDATE TIME: 17:44         PRINT TIME: 10:44     LANGUAGE: SPFORT      TYPE: CALLED
SECTION: SOURCE          UPDATE FOR: ENC            VERS/MODIF: 001/0001

ORG LINE    COL   1         2         3         4         5         6         7         8         9             S  IDENT
COL NBR   ----5----0----5----0----5----0----5----5----0----5----0----5----0----5----0----5----0----5             R  FIELD

 07     1          FUNCTION NAMER(A)                                                                                1*
 07     2          COMMON /INWORK/ NOBE,NOBLOK(20,6),NAME1(6),NDINV(20)                                             7*
 07     3          COMMON/CONST /LEK                                                                                8*
 07     4          COMMON/STATE/LABEL(6),LIST(1670),ISEL(6,20),NSTATE                                               9*
 08     5                  *LINBEG,LINEND,LENGTH,NENGTH,LPOINT,LEVEL,ITYPE,LTYPE                                    9*
 07     6         1K = 0                                                                                           11*
 07     7          I = LPOINT                                                                                      12*
 07     8          DO WHILE (I .LT. LINEND .AND. K .LT. 6)                                                         13*
 07     9             IF (LIST(I) .NE. LEK) THEN                                                                   14*
 07    10                K = K + 1                                                                                 15*
 07    11                NAME1(K) = LIST(I)                                                                        16*
 07    12             END IF                                                                                       17*
 07    13             I = I + 1                                                                                    18*
 07    14          END WHILE                                                                                       19*
 07    15          DO WHILE (K .LT. 6)                                                                             20*
 07    16             K = K + 1                                                                                    21*
 07    17             NAME1(K) = LEK                                                                               22*
 07    18          END WHILE                                                                                       23*
 07    19          NAMER = 0                                                                                       24*
 07    20          I = 1                                                                                           25*
 07    21          DO WHILE (I .LE. NOBE)                                                                          26*
 07    22             K = 1                                                                                        27*
 07    23             DO WHILE (NAME1(K ) .EQ. NOBLOK(I,K) .AND. K .LE. 6)                                         28*
 07    24                K = K + 1                                                                                 29*
 07    25             END WHILE                                                                                    30*
 07    26             IF (K .EQ. 7) THEN                                                                           31*
 07    27                NAMER = I                                                                                 32*
 07    28                I = NOBE                                                                                  33*
 07    29             END IF                                                                                       34*
 07    30             I = I + 1                                                                                    35*
 07    31          END WHILE                                                                                       36*
 07    32          RETURN                                                                                          37*
 07    33          END                                                                                             38*
```

Figure 3-14. Structured FORTRAN Listing

section-name - name of section.

unit-name - name of unit to be output; ALL indicates that all units in section are to be output.

MEDIA - default will produce a printed listing; TAPE and CARD will produce a direct copy of the data lines; TAPE requires a tape-card; see BACKUP Function for example of use of tape-card.

NBRLINES - number of lines of unit printed on one page (excluding header lines); value must be in range of 1 through 99 and must be expressed in one to six digits; default is 50.

INDENT - may be used to suppress automatic identation if language is structured; if unstructured, option is ignored.

The following example will list all the units in the JOB section.

```
** PARAM       PROJECT=FHACD147,LIBR=PROVEN
** SOURCE      SECT=JOB,UNIT=ALL
```

The following card, added to the example above, will list all the units in the SOURCE section, with automatic indentation where appropriate:

```
** SOURCE      SECT=SOURCE,UNIT=ALL
```

This Function should not be used in the same job as a BACKUP Function or a Terminate-with-BACKUP Function. See description of BACKUP Function.

If the Structured Programming Checking option was selected for the section when it was created and if one of these criteria is not met, an error flag will be inserted in the SP column on the unit listing. The codes used for the flag are as follows:

a. 1 = Unconditional transfer or address modification found on line.

b. 2 = SP length exceeded. See CREATE Function for parameter.

c. 3 = Both of the above were found on the line.

d. 4 = More than one statement on line.

e. 5 = Conditions 1 and 4 exist on line.

f. 6 = Conditions 2 and 4 exist on line.

g. 7 = All three error conditions on line.

3.4.28 TERMINATE Function

This Function is used to delete a section, a library, or an entire project. When file space is released by the TERMINATE Function, the FMS directive "PURGE" is used to overwrite the file space.

```
   ** TERMINATE   PROJECT=project-name,
   **             LIBRARY= {library-name} ,
                           {ALL         }
   **             SECTION= {section-name} ,
                           {ALL         }
 [ **             BACKUP=  { NO  } ]
 [                         { YES } ]
```

Plus a tape card with a file name of UT, if BACKUP=YES:

```
col
1
@ASG,C     UT,U9V,Z01530
```

This is a Exec 8 control card and follows the appropriate format. The tape card must be placed after the @END PSL card at the end of the PSL Function cards. See BACKUP for example.

The value for the TERMINATE parameters are:

project-name - name of project.

library-name - name of library; ALL will terminate entire project and purge the project index.

section-name - name of section; ALL will terminate entire library; not required and ignored if LIBRARY=ALL.

BACKUP - YES will produce a BACKUP tape of section(s) before deletion; default is NO.

The following example will delete the OBJECT section and save the section on a BACKUP tape:

```
Col
1
@USE            PSL.,DMA*PSL.
@ADD            PSL.RUN
** TERMINATE    PROJ=FHACD123,LIBR=MYLIB,
**              SECT=OBJECT,BACKUP=YES
@END            PSL
@ASG,C          UT,U9V,ZØ162Ø
@XQT            PSL.BCTL
```

Between the @ADD card and the @END card, the user may place an entire series of PSL Functions, but the user should be aware of the sequence in which output is written on the output tape, if BACKUP=YES. See description of BACKUP Function.

3.5 Sample Inputs

Sample inputs are included as examples in the descriptions of the individual PSL Functions in subsection 3.4. Appendix B shows a sample input stream which would build a user library of code and execute the user code.

3.6 Procedures for Output

One of the principal objectives of the PSL is to provide the exact status of a system to the programmers and to management. In order to provide a common, visible record of the developing system, an external (programmer-readable) version of the system is maintained in up-to-date correspondence with the internal (computer-readable) version. The files on disk constitute the internal library, and the corresponding listings, properly filed, make up the external library. See Figure 3-15. Standardized filing procedures have been established to ensure that the format of the external libraries is the same across programming projects.

The responsibility for the maintenance of the library should be assigned to one person. In order to make it possible for specially trained clerical personnel either to assume this responsibility or to work directly with a programmer who has the responsibility, the external procedures necessary to invoke the PSL Functions and to file the output have been expressly designed to be as straightforward as is practical. Recommended procedures for filing the output, in either section notebooks or in archives, are given below.

3.6.1 Section Notebooks

An up-to-date reference file is maintained for each section under a library. Each reference corresponds exactly to the section in the internal library. The actual physical form of storage for a section may vary, depending on the size and nature of the output generated for a unit in that section.

3.6.1.1 Card-Image Data

The units of the following sections usually contain card-image data:

a.	JOB	-	User's execution JCL
b.	LINK	-	Loader control cards
c.	PDL	-	Program Design Language data
d.	SOURCE	-	Source program data
e.	TEST	-	Test data
f.	TEXT	-	Documentation
g.	MGMT	-	Management data from manual input

One or more loose-leaf notebooks (11 x 16 1/2) is used for each of these sections, if the section has been created in the library. When units in these sections are created or updated,

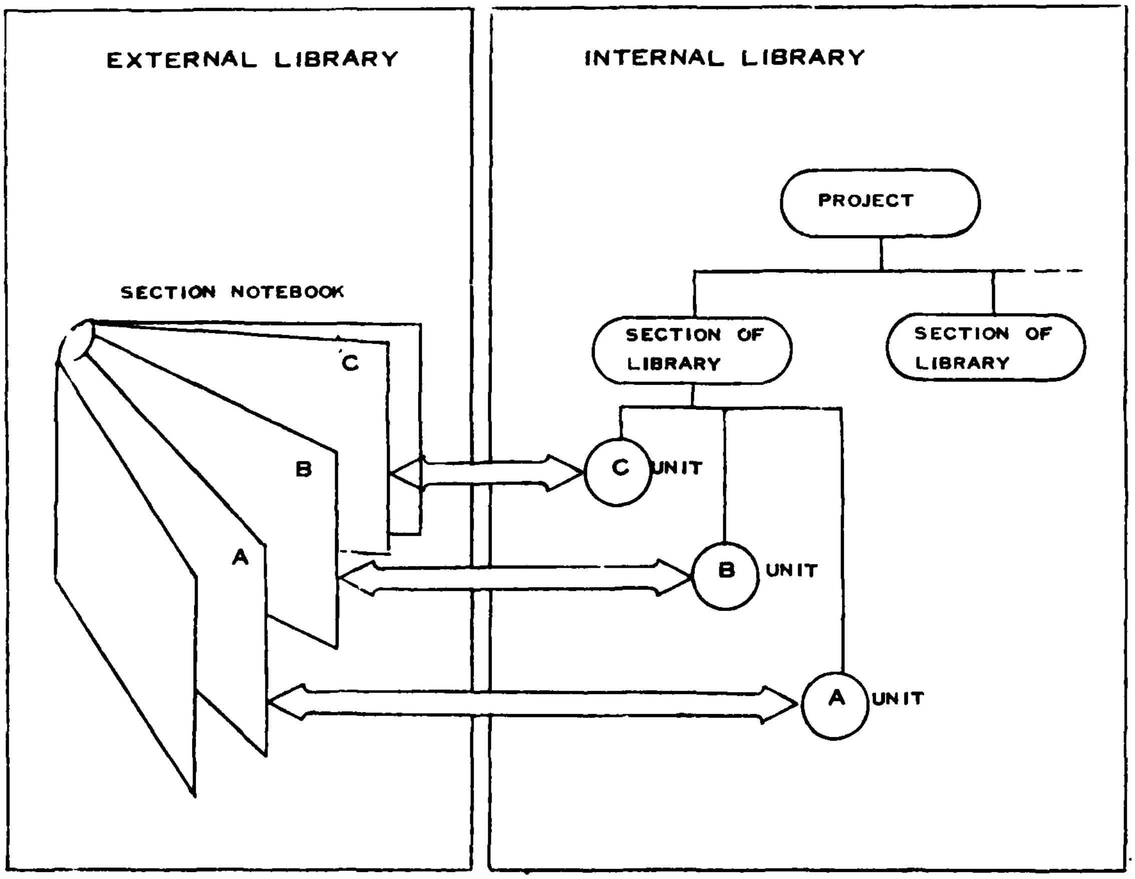


Figure 3-15. Internal and External Libraries

listings are provided automatically by the PSL system. The computer output is burst and the unit listing is filed in alphabetical order by unit name in the section notebook. An index listing, which contains the names and attributes of all of the units in the section, is filed at the beginning of the notebook. This index listing is regenerated as necessary with the INDEX Function to maintain a current and accurate picture of the section.

For the SOURCE and PDL sections, the MDPRINT Function may be used to obtain a Program Structure Report for the top units in the section. This report is used to determine the status of the top-down development of a program.

The Functions which affect the contents of the notebooks for these seven sections are:

a.	ADD	-	Add a unit
b.	REPLACE	-	Replace a unit
c.	CHANGE	-	Change a unit
d.	MOVE	-	Move a unit
e.	PURGE	-	Purge a unit
f.	SOURCE	-	Print a unit
g.	INDEX	-	Print a section index
h.	MDPRINT	-	Program Structure Report, Management Data Report, User Generated Report
i.	AUTHOR	-	Print specific programmer's units or index of units
j.	DOCUMENT	-	Print a document
k.	MDFORMAT	-	Maintain management data formats
l.	MDPLAN	-	Maintain management data plan
m.	MDUPDATE	-	Maintain management data
n.	MDXCHECK	-	Maintain management data exception checking

3.6.1.2 OBJECT Section

When the COMPILE Function is invoked, the compiler or assembler output is filed, unburst, in alphabetical order by OBJECT unit name. This is the unit-name of the top-most SOURCE unit, and is the unit name declared on the COMPILE Function card. This output may be filed in notebooks or post-binders, but in many projects the output is voluminous enough that it is more convenient to establish a file cabinet or slotted bookcase of suitable dimensions. An up-to-date index is also maintained.

The Functions which may affect the contents of the OBJECT section files are:

a. COMPILE

b. INDEX

3.6.1.3 LOAD Section

When the LINK Function is invoked, the Loader output listing is filed unburst in alphabetical order by LOAD unit name. This name is obtained from the LINK or OBJECT unit name, which was declared on the LINK card. The LOAD section is maintained in a loose-leaf notebook. A current index is filed at the beginning of the notebook. The Functions which may affect the content of the LOAD section notebook are:

a. LINK

b. INDEX

3.6.2 Archives

An archives box is maintained for each section. When a unit listing is replaced, the old listing is filed in the archives box. In addition, the computer output from a development project which does not correspond to one of the sections of the library is filed in one of the archives described below. All archives are placed unburst in chronological order in a notebook, post-binder, or box (most recent on top).

3.6.2.1 Library Maintenance Archives

Those PSL Functions which are used to initiate, terminate, and maintain the library are:

a. INITIAL - Initialize a project

b. CREATE - Create or change a section

c. TERMINATE - Terminate a section or sections

d. BACKUP - Backup

e. RESTORE - Restore

The listings from the runs which perform these Functions for a library are stored in the Library Maintenance Archives.

3.6.2.2 Execution Archives

Under the PSL system, a test run is performed with the EXECUTE Function. Additional control card information may also be added with the JCL Function. The entire listing from a test run is filed unburst in an Execution Archives box. A box is maintained for each program. The boxes are arranged on shelves in alphabetical order by JOB unit name. The Functions which produce output for Execution Archives are:

a. EXECUTE

b. JCL

APPENDIX A. STRUCTURE OF A SECTION

A section in a library under the PSL system is created in the format of a PSL Standard File. The organization of a section is shown in Figure A-01. The structure of the PSL Standard File is described below.

PSL Standard File Format

The PSL Standard File is a COBOL random (relative access) file. The file contains 128-word blocks (768 6-bit characters). A Standard PSL File will be created for each section, except PROGRAM, when the CREATE Function is invoked for the section. For a PROGRAM section the CREATE Function catalogs a program file for the storing of relocatable and absolute elements. The actual name of a section file is a concatenation of a one-character section code and the name of the user's library. Thus each section within each library, under a given project, is represented by a PSL Standard File. Figure A-02 lists the one character codes for each section.

The first block in a PSL Standard File is the first Control Block for the file. It contains information pertinent to the entire section, such as section name, user-selected options, file size, and the block flags which are used to control the space allotment within the file. The first Control Block also contains the block number of the first Index Block for the file.

The Index Blocks contain entries pointing to the locating of each data unit within the section. Initially, only one block is assigned to the index. When that block is full, another Index Block is developed and half of the entries from the first block are moved to the new block. In this way, a multi-level index is built as new data units are added.

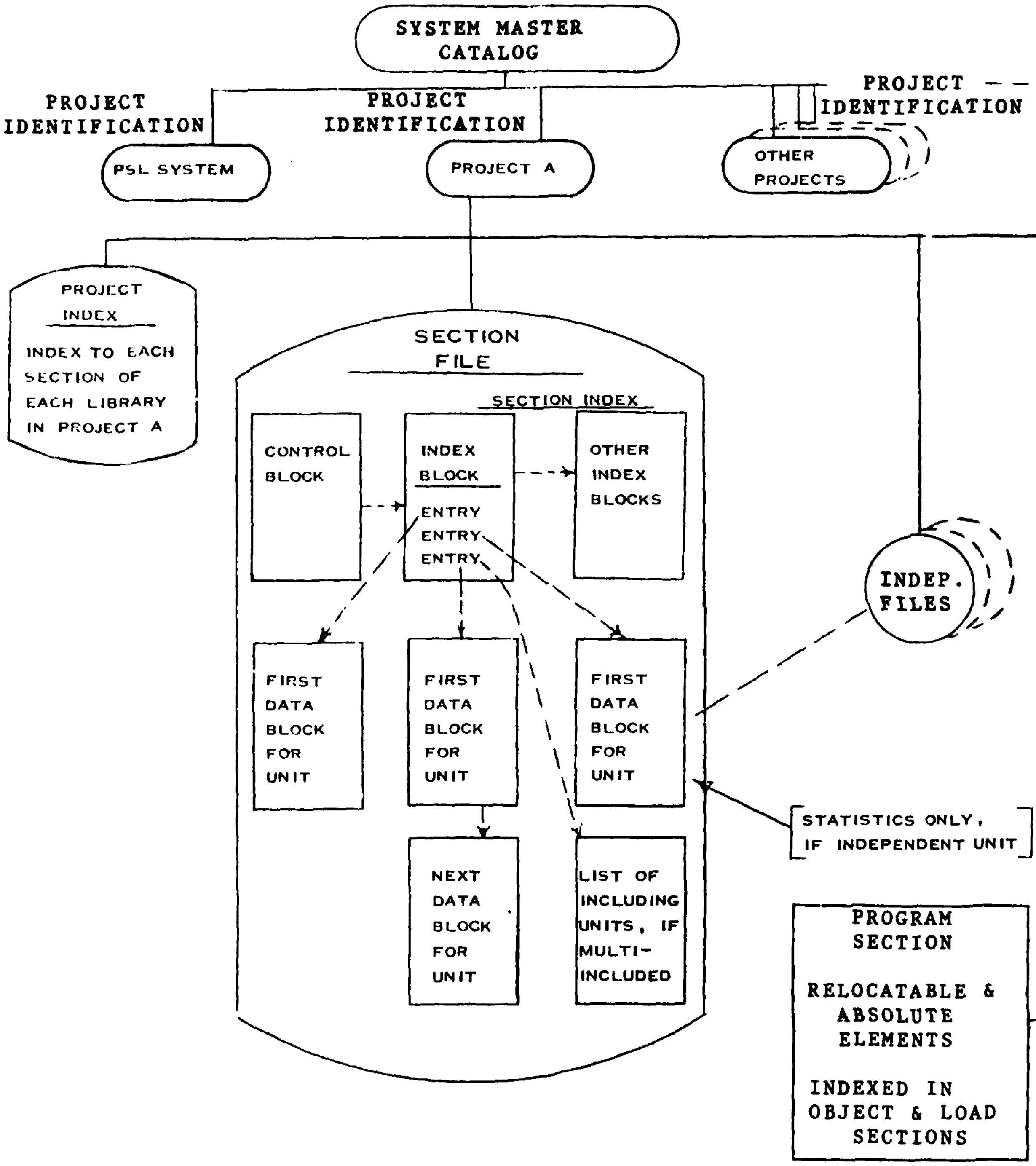

SYSTEM MASTER CATALOG
PROJECT IDENTIFICATION
PROJECT IDENTIFICATION
PROJECT
IDENTIFICATION
PSL SYSTEM
PROJECT A
OTHER PROJECTS
PROJECT INDEX
INDEX TO EACH SECTION OF EACH LIBRARY IN PROJECT A
SECTION FILE
SECTION INDEX
CONTROL BLOCK
INDEX BLOCK
ENTRY
ENTRY
ENTRY
OTHER INDEX BLOCKS
INDEP. FILES
FIRST DATA BLOCK FOR UNIT
FIRST DATA BLOCK FOR UNIT
FIRST DATA BLOCK FOR UNIT
STATISTICS ONLY, IF INDEPENDENT UNIT
NEXT DATA BLOCK FOR UNIT
LIST OF INCLUDING UNITS, IF MULTI-INCLUDED
PROGRAM SECTION
RELOCATABLE & ABSOLUTE ELEMENTS
INDEXED IN OBJECT & LOAD SECTIONS

Figure A-01. Organization of a PSL Sectior.

Section Name	Section Code
JOB	J
LINK	L
LOAD	X
OBJECT	O
PDL	P
SOURCE	S
TEST	D
TEXT	T
MGMT	M
USER	U
PROGRAM	Z

Figure A-02. Section Codes

A data unit within a section may be stored in blocks within the PSL Standard File or in a completely separate sequential file, referred to as an independent file. The section type and unit type determine which format is used for any given unit. Structured SOURCE, unstructured source code processed by the General Preprocessor (see Appendix I), PDL units and straightforward card image data are usually stored in blocks within the PSL Standard File. Unstructured SOURCE units are stored in independent files and must have unique names in the project. OBJECT and LOAD units are indexed and kept track of in the OBJECT and LOAD sections but the relocatable and absolute modules are stored in the PROGRAM section with element names equal to the unit names.

APPENDIX B. SAMPLE INPUT

The following pages of Appendix B contain sample input which illustrates the use of the PSL Functions.

```
•• PARAM PROJECT=FHACD149
•• INITIAL      PROJECT=FHACD149
•• INITIAL      PROJECT=FHACD147
•• PARAM        PROJECT=FHACD147,LIBRARY=PROVEN
•• CREATE SECT=PROG,FMS=(10,100)
•• CREATE SECTION=SOURCE,COMPRESS=YES,FMS=5
•• INDEX
•• CREATE SECTION=OBJECT,FMS=1
•• INDEX
•• CREATE SECTION=MGMT,FMS=15
•• INDEX
•• CREATE SECTION=LINK,FMS=1,STANDARD=YES
•• INDEX
•• CREATE SECTION=JOB,FMS=1
•• INDEX
•• CREATE SECTION=TEST,FMS=1
•• INDEX
•• PARAM        PROJECT=FHACD149,LIBRARY=NEWCODE
•• CREATE SECT=PROG,FMS=(10,100)
•• CREATE       SECTION=SOURCE,COMPRESS=YES,FMS=5,
••        MGMTDATA=YES,SPCHECK=YES,SPLENGTH=40
•• INDEX
•• CREATE SECTION=MGMT,FMS=20,COMPRESS=YES
•• INDEX
•• CREATE SECTION=PDL,COMPRESS=YES,FMS=1
•• INDEX
•• CREATE SECTION=TEXT,COMPRESS=YES,FMS=1
•• INDEX
•• CREATE       SECTION=OBJECT,PASSWORD=MAGIC,FMS=1
•• INDEX
```

2

```
** PARAM PROJECT=FHACD149
** PARAM        PROJECT=FHACD147,LIBRARY=PROVEN,SECTION=SOURCE
**  ADD     UNIT=TIPTOP,LANGUAGE=SCOBOL,PGMR=BERT
       IDENTIFICATION DIVISION.
         PROGRAM-ID.  TIPTOP.

           INCLUDE TIPTOP-ENVIRONMENT-DIVISION.

       DATA DIVISION.

           INCLUDE TIPTOP-FILE-SECTION.

           INCLUDE TIPTOP-WORKING-STORAGE.

       PROCEDURE DIVISION.

           INCLUDE TIPTOP-PROCEDURE-DIVISION.
**  ADD     UNIT=TIPTOP-ENVIRONMENT-DIVISION,PGMR=BERT
       ENVIRONMENT DIVISION.

        CONFIGURATION SECTION.

         SOURCE-COMPUTER.
           UNIVAC-1108.
         OBJECT-COMPUTER.
           UNIVAC-1108.

     *          SPECIAL-NAMES.
     *            PROCESS ALL DEBUG STATEMENTS.


        INPUT-OUTPUT SECTION.

         FILE-CONTROL.
           SELECT INPUT-CARDS
             ASSIGN TO CARD-READER CC.
           SELECT MESSAGE-FILE
               ASSIGN TO PRINTER MS.
         I-O-CONTROL.
           APPLY EXPEE ON INPUT-CARDS,
               MESSAGE-FILE.
**  ADD     UNIT=TIPTOP-FILE-SECTION,PGMR=BERT
        FILE SECTION.



        FD  INPUT-CARDS
               LABEL RECORDS ARE OMITTED.

         01  INPUT-CARD                   PIC X(80).
```

```
           FD  MESSAGE-FILE
                   LABEL RECORDS ARE OMITTED.

            01  MESSAGE-LINE                  PIC X(120).

** ADD SECTION=LINK,UNIT=TIPTOP,PGMR=CHARLES
         INCLUDE TIPTOP
         INCLUDE ADUNE
         INCLUDE CRSC
         INCLUDE ITPJ
         INCLUDE OPEN
         INCLUDE OPUS
         INCLUDE PRMS
         INCLUDE OPKW
         INCLUDE OPFD
INCLUDE FRUN
 INCLUDE FLAF
** ADD SECTION=JOB,UNIT=TIPTOP,UTYPE=INDEP,PGMR=CHARLES,
** FMS=1
@ASG,CP MC,F///100
@ASG,A MS,F///100
@ASG,T JS,F///100
@ASG,T UC,F///100
@ASG,CP UL,F///100
@ASG,A UL,F///100
@ASG,T PC,F///100
@ASG,T PK,F///100
@ASG,T FC,F///100
@ASG,T NC,F///100
@ASG,T UT,F///100
@ASG,T FJ,F///100
@ASG,T FS,F///100
@ASG,T TF,F///100
@ASG,T CHF,F///10
@ASG,T XF,F///100
@ASG,T XL,F///100
@ASG:T NF,F///100
@ASG,T CF,F///100
@ASG,T PT,F///100
@ASG,T XA,F///100
@ASG,T F1,F///100
@ASG,T F2,F///100
@ASG,T F3,F///100
@ASG,T R0,F///100
@ASG,T R1,F///100
@ASG,T R2,F///100
@ASG,T R3,F///100
@ASG,T R4,F///100
@DATA,I CC... PSL
@END PSL
@XQT                                                               EXECUTE
** PARAM       PROJECT=FHACD149,LIBRARY=NEWCODE,SECTION=SOURCE
**  ADD     UNIT=TIPTOP-PROCEDURE-DIVISION,UTYPE=INCL,PGMR=CHARLES,
**        LANGUAGE=SCOBOL
          PATCH-PSL-CONTROL.
      1    DISPLAY ' TRACE - PCTL (PATCH-PSL-CONTROL) EXECUTED.'.
```

```
** PARAM PROJ=FHACD149,LIBRARY=NEWCODE,SECTION=PDL,PGMR=BERT
** ADD UNIT=TIPTOP-PDL
     *     THE MODULE TIPTOP PROCESSES THE ADD A UNIT FUNCTION (ADUN)

        TIPTOP
            INITIALIZE PARAMETER-TABLE WITH SPACES
            CALL 'OBUS'
            MOVE PGMR-NAME TO PARAMETER-TABLE
            OPEN INPUT INPUT-CARDS,
                  OUTPUT MESSAGE-FILE
            MOVE 'ADD' TO INPUT-FUNCTION
            READ INPUT-CARDS
            IF NOT END OF INPUT CARDS
              CALL 'OBFN'
              DO UNTIL NORMAL RETURN FROM OBFN
                SEARCH PSL-FUNCTIONS
                  WHEN PSL-FUNCTION IN THE TABLE EQUAL INPUT-FUNCTION
                  SET FUNCTION-NUMBER TO INDEX OF TABLE.
                INCLUDE PROCESS-FUNCTION.
                CALL 'OBFN'
              ENDDO.
              CALL PRINT MESSAGE (END OF INPUT CARDS).
              IF SPAWN JOB NEEDED
                INCLUDE SPAWN-A-JOB
              ENDIF.
            ELSE
              CALL PRINT MESSAGE (NO INPUT CARDS)
            ENDIF.
            CALL PLAF.
            CLOSE INPUT-CARDS,
                  MESSAGE-FILE.
            STOP RUN.
** PARAM PROJ=FHACD149,LIBRARY=NEWCODE,SECTION=TEXT,PGMR=CHARLES
** ADD UNIT=TIPTOP-TEXT
*******
        THE MODULE TIPTOP PROCESSES THE PSL FUNCTION,ADD A UNIT (ADUN)
        NEW UNITS ARE ADDED TO A SPECIFIED PSL LIBRARY OR CODE ADDED TO
        STUP UNITS.
        A LISTING OF THE ADDED UNIT IS PROVIDED.
*******
** INDEX        PROJECT=FHACD147,LIBRARY=PROVEN,SECT=SOURCE
** INDEX        PROJECT=FHACD149,LIBRARY=NEWCODE,SECT=SOURCE
** PARAM    PROJECT=FHACD149,LIBRARY=NEWCODE
** DOCUMENT  LSPACE=2,PAGE=0,SECTION=TEXT
**   HEADER HSPACE=3
                            TIPTOP-TEXT-HEADER
** TEXT UNIT=TIPTOP-TEXT
** DOCUMENT  PAGE=1,SECTION=PDL
**   HEADER  HSPACE=10
                            TIPTOP-PDL-HEADER
**   TEXT UNIT=TIPTOP-PDL
** EJECT
** TEXT
            THIS TEXT SHOULD BE AT THE TOP OF THE PAGE.
```

```
** PARAM PROJECT=FHACD149
** PARAM PROJECT=FHACD147,LIBRARY=PROVEN,SECTION=MGMT
** MFFORMAT LEVEL=SYSTEM,OP=ADD
I010      12PROJ-TITLE    PROJECT TITLE
I020 RPT 4FPROJ-DESCR    PROJECT DESCRIPTION
I030     NO6START-DATE   PROJECT START DATE
I040     NO6EST-END-DATE ESTIMATED COMPLETION DATE
I050     NO6ACT-END-DATE ACTUAL COMPLETION DATE
I060     NO3P-AVE-MGRS   PLANNED AVERAGE YEARS EXPERIENCE MANAGERS
I070     NO3P-AVE-ANAL   PLANNED AVERAGE YEARS EXPERIENCE ANALYSTS
I080     NO3P-AVE-PROG   PLANNED AVERAGE YEARS EXPERIENCE PROGRAMMERS
I090     NO3P-AVE-ADMIN  PLANNED AVERAGE YEARS EXPERIENCE ADMINISTRATIVE
I100     NO3P-AVE-OTH    PLANNED AVERAGE YEARS EXPERIENCE OTHER
I110     NO3A-AVE-MGRS   ACTUAL AVERAGE YEARS EXPERIENCE MANAGERS
I120     NO3A-AVE-ANAL   ACTUAL AVERAGE YEARS EXPERIENCE ANALYSTS
I130     NO3A-AVE-PROG   ACTUAL AVERAGE YEARS EXPERIENCE PROGRAMMERS
I140     NO3A-AVE-ADMIN  ACTUAL AVERAGE YEARS EXPERIENCE ADMINISTRATIVE
I150     NO3A-AVE-OTH    ACTUAL AVERAGE YEARS EXPERIENCE OTHER
I160     NO4P-NBR-MGRS   PLANNED NUMBER OF MANAGERS
I170     NO4P-NBR-ANAL   PLANNED NUMBER OF ANALYSTS
I180     NO4P-NBR-PROG   PLANNED NUMBER OF PROGRAMMERS
I190     NO4P-NBR-ADMIN  PLANNED NUMBER OF ADMINISTRATIVE
I200     NO4P-NBR-OTH    PLANNED NUMBER OF OTHER
I210     NO4A-NBR-MGRS   ACTUAL NUMBER OF MANAGERS
I220     NO4A-NBR-ANAL   ACTUAL NUMBER OF ANALYSTS
I230     NO4A-NBR-PROG   ACTUAL NUMBER OF PROGRAMMERS
I240     NO4A-NBR-ADMIN  ACTUAL NUMBER OF ADMINISTRATIVE
I250     NO4A-NBR-OTH    ACTUAL NUMBER OF OTHER
I260     NO3E-TURNOVER   ESTIMATED PERSONNEL TURNOVER RATE
I270     NO3A-TURNOVER   ACTUAL PERSONNEL TURNOVER RATE
I280     NO3E-LOC-TRIPS  ESTIMATED LOCAL TRAVEL
I290     NO3A-LOC-TRIPS  ACTUAL LOCAL TRAVEL
I300     NO3E-DIS-TRIPS  ESTIMATED DISTANT TRAVEL
I310     NO3A-DIS-TRIPS  ACTUAL DISTANT TRAVEL
I320     NO1E-WORK-COND  ESTIMATED WORKING CONDITIONS
I330     NO1A-WORK-COND  ACTUAL WORKING CONDITIONS
I340     NO2P-LANG-EXP   PLANNED PROGRAMMING LANGUAGE EXPERIENCE
I350     NO2A-LANG-EXP   ACTUAL PROGRAMMING LANGUAGE EXPERIENCE
I360     NO2P-SIM-EXP    PLANNED SIMILAR APPLICATION EXPERIENCE
I370     NO2A-SIM-EXP    ACTUAL SIMILAR APPLICATION EXPERIENCE
I380     NO2P-TARG-EXP   PLANNED TARGET COMPUTER EXPERIENCE
I390     NO2A-TARG-EXP   ACTUAL TARGET COMPUTER EXPERIENCE
I400     NO2E-APPL-EXP   ESTIMATED CUSTOMER APPLICATION EXPERIENCE
I410     NO2A-APPL-EXP   ACTUAL CUSTOMER APPLICATION EXPERIENCE
I420     NO2E-EQUIP-EXP  ESTIMATED CUSTOMER EQUIPMENT EXPERIENCE
I430     NO2A-EQUIP-EXP  ACTUAL CUSTOMER EQUIPMENT EXPERIENCE
I440     NO1E-COMPLEXITY ESTMATED COMPLEXITY OF PROJECT
I450     NO1A-COMPLEXITY ACTUAL COMPLEXITY OF PROJECT
I460     NO4EDOC-FUNC-SP ESTMATED PAGES DOCUMENTATION FUNCTIONAL SPECS
I470     NO4EDOC-USER-GU ESTMATED PAGES DOCUMENTATION USERS GUIDE
I480     NO4EDOC-TEST-SP ESTMATED PAGES DOCUMENTATION TEST SPECIFICATIONS
I490     NO4EDOC-PROG-LT ESTIMATED PAGES DOCUMENTATION PROGRAM LISTINGS
I500     NO4ADOC-FUNC-SP ACTUAL PAGES DOCUMENTATION FUNCTIONAL SPECS
I510     NO4ADOC-USER-GU ACTUAL PAGES DOCUMENTATION USERS GUIDE
```

```
** PARAM PROJECT=FHACD147,LIBRARY=NEWCODE
** MDFORMAT LEVEL=SYSTEM,OP=MOVE,OLDP=FHACD147,OLDL=PROVEN
** MDFORMAT LEVEL=SUBSYSTEM,OP=MOVE,OLDP=FHACD147,OLDL=PROVEN
** MDFORMAT LEVEL=MODULE,OP=MOVE,OLDP=FHACD147,OLDL=PROVEN
** MDFORMAT LEVEL=JOB,OP=MOVE,OLDP=FHACD147,OLDL=PROVEN
** MDFORMAT LEVEL=UNIT,OP=MOVE,OLDP=FHACD147,OLDL=PROVEN
** MDPLAN MOD=0
**    INSERT AFTER=0
          SYSTEM=NEWCODE,LIB2=PROVEN,LIB3=JOHN,
             PROJ2=FHACD147,PROJ3=FHACD149

             SUBSYS=NEWCODE-PM
                MODULE=TIPTOP
                MODULE=ADUNE

             SUBSYS=NEWCODE-CU
                JOB=TIPTOP-CU
                JOB=ADUN-CU
** PARAM PROJECT=FHACD147,LIBRARY=PROVEN
** MDUPDATE UNIT=PROVEN,LEVEL=SYSTEM,OP=ADD
010=PROVEN-TITLE
020=PROVEN TEST PROJECT DESCRIPTION
030=770601
040=770801
050=770810
060=008
070=005
080=003
090=002
100=001
110=010
120=008
130=003
140=003
150=002
160=0001
070=0003
080=0005
090=0002
200=0001
210=0001
220=0002
230=0002
240=0001
250=0001
260=001
270=002
280=040
290=048
300=010
310=008
320=3
330=3
340=03
350=04
```

```
•• TERMINATE     PROJECT=FHACD149.LIBRARY=MARY.SECTION=SOURCE
•• CREATE FMS=3
•• ADD           UNIT=INTERPRET-KEYWORD-VALUES.UTYPE=INCL.KEY=MYSTERY.
••               LANG=SCOBOL
             INTERPRET-KEYWORD-VALUES.
        3    DISPLAY ' TRACE - ADUN (INTERPRET-KEYWORD-VALUES) EXECUTED.'.
             CASENTRY KEYWORD-NBR
               CASE 2
                 IF LENGTH-OF-VALUE GREATER THAN LENGTH-OF-KEY-VALUE
                   MOVE CODE-FOR-VALUE-ERROR TO RETURN-CODE
                 ELSE
                   MOVE KEYWORD-VALUE TO INPUT-UNIT-KEY
                 ENDIF
               CASE 3
                 IF LENGTH-OF-VALUE GREATER THAN LENGTH-OF-LANGUAGE-NAME
                   MOVE CODE-FOR-VALUE-ERROR TO RETURN-CODE
                 ELSE
                   MOVE KEYWORD-VALUE TO INPUT-UNIT-LANGUAGE
                 ENDIF
               CASE4
                 MOVE SPACES TO LIBRARY-NAME OF PARAMETER-TABLE
                 IF LENGTH-OF-VALUE GREATER THAN LENGTH-OF-LIBRARY-NAME
                   MOVE CODE-FOR-VALUE-ERROR TO RETURN-CODE
                 ELSE
                   MOVE KEYWORD-VALUE TO
                         LIBRARY-NAME OF PARAMETER-TABLE
                 ENDIF
             ENDCASE
   INTERPRET-KEYWORD-VALUES.
             EXIT.
•• CHANGE        UNIT=INTERPRET-KEYWORD-VALUES.KEY=MYSTERY
•• SHIFT LINE=1.COLU=L4
•• MODIFY        LINES=(5.9)
                 MOVE SPACE TO SECTION-CODE OF PARAMETER-TABLE
                 IF LENGTH-OF-VALUE GREATER THAN LENGTH-OF-SECTION-NAME
                   MOVE CODE-FOR-VALUE-ERROR TO RETURN-CODE
                 ELSE
                   MOVE KEYWORD-VALUE TO INPUT-SECTION-NAME
                   SET ST-INDEX TO 1
                   SEARCH SECTION-TABLE-ENTRIES
                       AT END
                         MOVE CODE-FOR-VALUE-ERROR TO RETURN-CODE
                       WHEN SECTION-TABLE-NAME (ST-INDEX)
                             EQUAL TO INPUT-SECTION-NAME
                         MOVE SECTION-TABLE-NAME-CODE (ST-INDEX)
                               TO SECTION-CODE OF PARAMETER-TABLE.
                 ENDIF
•• DELETE        LINES=(10.15)
•• MODIFY L=16.FROM=CASE4.TO='CASE 4'
•• INSERT        AFTER=23
               CASE 5
                 IF LENGTH-OF-VALUE GREATER THAN LENGTH-OF-PASSWORD
                   MOVE SPACES TO PASSWORD OF PARAMETER-TABLE
                   MOVE CODE-FOR-VALUE-ERROR TO RETURN-CODE
                 ELSE
                   MOVE KEYWORD-VALUE TO PASSWORD OF PARAMETER-TABLE.
                 ENDIF
•• COPY AFTER=24.OLDUNIT=ADUN-PROCEDURE-DIVISION.OLDL=JOHN.
••      FROM=16
•• SHIFT LINE=25.COLU=R4
•• MODIFY LINE=25.COLU=32.TO=-EXIT.
•• PARAM    PROJECT=FHACD147•LIBRARY=PROVFN•SECTION=SOURCE
•• CSCAN    STRINC=TIPTOP
•• PARAM          PROJECT=FHACD149•LIBRARY=JOHN•SECTION=LOAD
•• CREATE         FMS=1
•• LINK           LINK=TIPTOP•LIF2=NEWCODE•LIB3=PROVEN•PROJ3=FHACD147
•• EXFCUTE        JOB=TIPTOP•PROJ2=FHACD147•LIB2=PROVEN
•• INDEX
•• PARAM PROJECT=FHACD147•LIBRARY=PROVEN•SECTION=MGMT
•• MDPLAN MOD=0
••  INSERT AFTER=0
    SYSTEM=PROVEN•LIB1=PROVEN
      SUBSYS=SUBTOP
      MODULE=TIPTOP•LIB2=NEWCODE•PROJ2=FHACD149
      MODULE=ADUNI•LIB3=JOHN
        JOB=TIPTOP-CU
        JOB=ADUN-CU
•• PARAM PROJECT=FHACD149•LIBRARY=NEWCODE
•• MDFORMAT LEVEL=SYSTEM•OP=CHANGE
R280    NO3E-LOC-TRIPCH ESTIMATED LOCL TRAVEL CHANGE
R450    NO1A-COMPLEX-CH ACTUAL COMPLEXITY OF PROJECT CHANGE
D710
IP30M210TTLLINES-COPIED LINES OF SOURCE CODE COPIED
IR40M220TTLREAL-UNIT-CT REAL UNIT COUNT
IR50M230TTLSTUB-UNIT-CT STUB UNIT COUNT
I860    NO4NBR-LINES-PL NUMBER OF SOURCE LINES PLANNED
R540      24SYS-NAMERPL SYSTEM NAME EDIT NBR REPLACED
R550      24SUBSYS-NAME  SUBSYSTEM NAME EDIT NBR REPLACDE
•• MDUPDATE UNIT=NEWCODE-PM•OP=CHANGE
•• INSERT
 260=00010000
•• MODIFY
 170=0075
 250=00001025
•• DELETE
 290
•• MDXCHECK UNIT=NEWCODE
•• INSERT
 210>160:V01•051177•MORE MGRS THAN PLANNED
 290>280:V10•051177•MORE LOC TRIPS THAN EST
 700>690:V50•051277•TOO FEW SUCCESSFUL RUNS
 730<860:V10•053077•TOO LITTLE SOURCE CODE
•• PARAM PROJECT=FHACD147•LIBRARY-PROVEN
•• MDXCHECK UNIT=PROVEN
•• INSERT
 450<440:V15•051377•COMPLEXITY OVER EST
 050>040:V00•051377•COMPLETION DATE NOT
 230>180:V01•051377•TOO MANY PROGRAMMERS
•• MDXCHECK UNIT=PROVEN
•• MODIFY
 450<440:V50•051777•COMPLEXITY OVER EST
•• INDEX
•• INDEX PROJECT=FHACD149•LIBRARY=NEWCODE
•• PARAM PROJECT=FHACD149•LIBRARY-NEWCODE
•• MDCOLLECT
•• MDPRINT REPORT=MD
•• PARAM PROJECT=FHACD147•LIBRARY-PROVEN
•• MDCOLLECT
•• MDPRINT REPORT-MD
```

```
** SOURCE LIBR=JOHN,SECT=SOURCE,
**  UNIT=ADUN-PROCEDURE-DIVISION,INDENT=NO
** AUTHOR LIBR=NEWCODE,UPGMR=CHARLES,OPGMR=CHARLES
** AUTHOR LIBR=PROVEN,UPGMR=BERT,PROJ=FHACD147,OPGMR=BERT
** AUTHOR LIBR=JOHN,UPGMR=WENDY,PROJ=FHACD149,OPGMR=WENDY
** COMPILE       UNIT=ADUNE,PROJECT=FHACD149,LIB1=JOHN,LIB2=NEWCODE,
**               OBJECT=YES
** EXECUTE       JOB=TIPTOP,LINK=TIPTOP,PROJ1=FHACD149,
**               LIB1=JOHN,LIB2=NEWCODE,
**               PROJ3=FHACD147,LIB3=PROVEN
** RESTORE      PROJ=FHACD149,LIBR=JOHN,UNIT=ADUN-PROCEDURE-DIVISION,
**              SECTION=SOURCE
** PARAM PROJ=FHACD149,SECT=SOURCE
** REPLACE      UNIT=STORE-INPUT-KEYWORDS,LIBRARY=JOHN
            MOVE SPACE TO SECTION-CODE OF PARAMETER-TABLE
            IF LENGTH-OF-VALUE GREATER THAN LENGTH-OF-SECTION-NAME
              MOVE CODE-FOR-VALUE-ERROR TO RETURN-CODE
            ELSE
              MOVE KEYWORD-VALUE TO INPUT-SECTION-NAME
              SET ST-INDEX TO 1
              SEARCH SECTION-TABLE-ENTRIES
                  AT END
                    MOVE CODE-FOR-VALUE-ERROR TO RETURN-CODE
                  WHEN SECTION-TABLE-NAME (ST-INDEX)
                       EQUAL TO INPUT-SECTION-NAME
                    MOVE SECTION-TABLE-NAME-CODE (ST-INDEX)
                         TO SECTION-CODE OF PARAMETER-TABLE.
            ENDIF
** SOURCE     UNIT=ADUN-PROCEDURE-DIVISION,NBRLINES=30,INDENT=NO
** SOURCE     UNIT=ALL
** PARAM PROJECT=FHACD149,LIBRARY=NEWCODE
** DOCUMENT LSPACE=3,PAGE=1,SECTION=TEXT
**   HEADER HSPACE=5
            TIPTOP DESCRIPTION
**   TEXT UNIT=TIPTOP-TEXT
** MDPLAN SECT=MGMT,MOD=1
**   DELETE LINE=5
** MDCOLLECT ARCHIVE=YES
**  MDCOLLECT ARCHIVE=YES
** MDPRINT REPORT=MD
** MDPRINT REPORT=MD,HIST=ALL
** TERMINATE PROJECT=FHACD149,LIBR=JOHN,SECT=ALL
** CREATE        SECT=SOURCE,MGMTDATA=YES,SPCHECK=YES,SPLENGTH=55,
**               FMS=5
** CREATE        SECT=OBJECT,FMS=1
** CREATE        SECT=LOAD,FMS=1
**  RESTORE      PROJECT=FHACD149,LIBR=ALL
**  PARAM        PROJECT=FHACD149,LIBRARY=NEWCODE,SECTION=SOURCE
**  MOVE         UNIT=ADUN-PROCEDURE-DIVISION,OLDLIB=JOHN,
**  REPLACE=YES
**  MOVE         UNIT=ADUN-WORKING-STORAGE-SECTION,OLDLIB=JOHN,
**  REPLACE=YES
**  MDPRINT      UNIT=ADUNE,REPORT=PS
**  COMPILE      UNIT=ADUNE,OPTIONS=TL,PASS=MAGIC,OBJECT=NO
**  PARAM        LIBRARY=JOHN,PASS=' '
**  PURGE        UNIT=ADUN-PROCEDURE-DIVISION
**  PURGE        UNIT=ADUN-WORKING-STORAGE-SECTION
**  INDEX
```

APPENDIX C. MESSAGE OUTPUT

Messages which are printed by the PSL system are divided into the following categories:

a. Advisory - A condition exists which may require user attention.

b. Error - An error has been encountered which has prevented normal completion of the processing.

c. FMS - The Executive System has returned an error status code. The status bit number (FMS-ERROR-NUMBER) and the appropriate error message (FMS-ERROR-MESSAGE) is printed by the PSL system. A subsequent message will give further information as to the effect of the error on the processing. For further detail of error messages, see Univac 11C0 Series Executive System Diagnostic Messages and Status Codes (Facility Status Bits Table).

d. Information - These messages provide status information to the user. They are printed during normal processing.

e. MSG - General messages which may be generated during execution of a PSL-spawned job.

f. PSL - A message in this category may indicate a problem in the PSL system. If such a condition should arise, the user should contact system support personnel.

In the following list, the messages are grouped according to category, which is indicated by the first three characters of the message numbet.

Msg. Key	Message Text

ADV006	UNABLE TO ASSIGN REQUESTED FILE WITH
	ACCESS=access-mode
	ALLOCATION=allocation-type
	PROJECT=project-name
	LIBRARY=library-name
	SECTION=section-name
	UNIT=unit-name

ADV016	JOB SPAWNED.
ADV054	PREVIOUSLY COLLECTED ELEMENT unit-name IN PLAN LINE line-nbr
ADV057	SUBORDINATE ELEMENT unit-name NOT PROCESSED IN PLAN LINE line-nbr
ADV058	UNABLE TO UPDATE BEYOND LAST LINE (last-line-nbr) IN UNIT
ADV070	REAL UNIT unit-name NO LONGER INCLUDED BY ANY HIGHER-LEVEL UNIT
ADV106	UNABLE TO MOVE UNIT=unit-name BECAUSE REPLACE=NO
ADV133	UNABLE TO MOVE STUBUNIT=unit-name

ADV137	RESTORE INITIATED FOR
	PROJECT=project-name LIBRARY=library-name
	SECTION=section-name

ADV140	RESTORE COMPLETED FOR
	PROJECT=project-name LIBRARY=library-name
	SECTION=section-name

ADV141	EXISTING UNIT DELETED PRIOR TO RESTORE UNIT=unit-name
ADV142	UNIT RESTORE COMPLETED FOR UNIT=unit-name
ADV150	UNABLE TO COPY BEYOND LINE line-nbr FOR UNIT unit-name
ADV160	STRING REPLACED repeat-nbr TIMES character-string
ERR001	INVALID FUNCTION function-name

Msg. Key	Message Text
ERR002	INVALID KEYWORD keyword
ERR003	ATTEMPTED TO ADD UNIT WHICH ALREADY EXISTS unit-name
ERR004	UNIT TYPE OF EXISTING UNIT unit-name IS unit-type. INCONSISTENT WITH REQUESTED UNIT TYPE.
ERR005	NO SPACE AVAILABLE UNDER LIBRARY library-name SECTION section-name
ERR007	UNABLE TO PROCESS PREVIOUS REQUESTED FUNCTION
ERR008	ATTEMPTED TO CHANGE STUB OR NON-EXISTENT UNIT unit-name
ERR009	INVALID UNIT KEY unit-key
ERR010	INVALID SECTION=section-name FOR REQUESTED FUNCTION
ERR011	INTERNAL TABLE-table-name EXCEEDED DUE TO NUMBER OF ITEMS. MAX. ENTRIES number-of-max-items.
ERR013	SYNTAX ERROR IN COLUMN column-nbr Note: Special mark indicates column in error.
ERR015	NO INPUT CARDS FOUND
ERR017	INVALID PLAN LINE line-nbr
ERR019	KEYWORD VALUE ERROR value
ERR023	UNABLE TO INITIALIZE STANDARD PSL FILE PROJECT=project-name LIBRARY=library-name SECTION=section-name
ERR026	UNIT unit-name NOT FOUND
ERR030	INVALID DATA. REJECTED MGMT UNIT - unit-name
ERR032	MISSING REQUIRED KEYWORD keyword
ERR035	ERROR ON PARAM CARD (SKIPPING TO NEXT PARAM CARD)

Msg. Key	Message Text
ERR036	SECTION ALREADY EXISTS IN LIBRARY
ERR053	LEVEL INCONSISTENT WITH FIRST OCCURANCE OF ELEMENT unit-name REPEATED ELEMENT REJECTED.
ERR055	REJECTED INCLUDE STATEMENT FOR UNIT unit-name
ERR056	MAXIMUM NUMBER OF CHARACTER (max-nbr) EXCEEDED FOR NAME OF UNIT unit-name
ERR059	MAXIMUM NUMBER OF CHANGES PER LINE = 6. LIMIT EXCEEDED FOR LINE NUMBER line-nbr. FIRST SIX CHANGES APPLIED.
ERR060	UNACCEPTABLE FMS DIRECTIVE fms-keyword
ERR061	INVALID FMS PERMISSION fms-entry
ERR062	MAXIMUM NUMBER OF FMS USER NAMES = 8. LIMIT EXCEEDED. EXCESS IGNORED.
ERR063	INVALID FMS DEVICE SPECIFIED fms-value
ERR064	INVALID REFERENCE BY level FORMAT, ITEM item-nbr TO level FORMAT, ITEM item-nbr
ERR065	INVALID XCHECK REFERENCE TO ITEM item-nbr IN MD-UNIT unit-name XCHECK SPECIFICATION xcheck-spec.
ERR066	ERROR WHILE PRINTING UNIT=unit-name
ERR068	INVALID UNIT TYPE (unit-type) FOR INCLUDED UNIT unit-name
ERR069	UNABLE TO PROCESS FMS FILE OPTIONS
ERR071	MISSING CONTINUATION CARD
ERR075	UNABLE TO ACCESS PROJECT=project-name LIBRARY=library-name SECTION=section-name UNIT=unit-name
ERR076	USER DATA REQUIRED

Msg. Key	Message Text
ERR081	PROCEDURE procedure-name NOT SUPPORTED
ERR082	INVALID VALUE FORMAT FOR KEYWORD (keyword)
ERR083	INVALID FLAGWORD (flagword-specification)
ERR084	INVALID ACCESS CODE (flagword-specification)
ERR085	INDEPENDENT FILE INVALID FOR STANDARD SECTION (section-name)
ERR088	INDEPENDENT SOURCE UNIT REQUIRED FOR PROCEDURES procedure-name
ERR092	ILLEGAL UNIT TYPE FOR COMPILE FUNCTION FOR UNIT=unit-name UNIT TYPE unit-type
ERR097	INVALID SECTION PASSWORD password PROJECT=project-name LIBRARY=library-name SECTION=section-name UNIT=unit-name
ERR098	USERID DOES NOT MATCH PROJECT
ERR100	INVALID MODIFICATION UNIT MODIFICATION IS CURRENTLY modif-level
ERR102	UNABLE TO MOVE UNIT=unit-name BECAUSE OF EXISTING UNIT-KEY
ERR103	ILLEGAL MOVE. SAME LIBRARY SPECIFIED.
ERR104	ILLEGAL MOVE. SAME UNIT SPECIFIED.
ERR105	UNABLE TO MOVE UNIT=unit-name
ERR131	UNABLE TO BACKUP LIBRARY=library-name SECTION=section-name
ERR134	NO BACKUP UNITS FOUND TO MEET RESTORE REQUIREMENTS
ERR135	THE FOLLOWING KEYWORD IS UNNECESSARY AND IN ERROR keyword
ERR136	RESTORE PROJECT UNMATCHED BY BACKUP PROJECT=project-name LIBRARY=library-name SECTION=section-name

Msg. Key	Message Text
ERR138	EXISTING SECTION TO BE RESTORED NOT FREE OF UNITS PROJECT=project-name LIBRARY=library-name SECTION=section-name
ERR143	UNIT RESTORE FAILED FOR UNIT=unit-name
ERR144	BACKUP END-OF-FILE ENCOUNTERED PREMATURELY
ERR145	BACKUP FILE FORMAT IN ERROR
ERR146	SECTION-HEADER MISSING ON BACKUP FILE
ERR147	SECTION RESTORE FAILED FOR PROJECT=project-name LIBRARY=library-name SECTION=section-name
ERR148	MAXIMUM NUMBER OF HEADER LINES (max.-header-line-count) EXCEEDED
ERR152	UNABLE TO OBTAIN FMS PARAMETERS FOR INDEP-UNIT PROJECT=project-name LIBRARY=library-name SECTION=section-name UNIT=unit-name
ERR153	UNABLE TO BACKUP UNIT unit-name LIBRARY=library-name SECTION=section-name
ERR155	OPGMR DOES NOT MATCH UPGMR
ERR158	RESERVED UNIT NAME SPECIFIED
ERR161	INVALID OP-CODE FOR RECORD data-record
ERR190	FORMAT ERROR edit-pointers
ERR193	ITEM NOT DESCRIBED IN APPROPRIATE FORMAT UNIT
ERR194	ITEM IS NOT MANUAL DATA
ERR195	ILLEGAL SPECIFICATION FOR XCHECK
ERR196	INVALID DATE FOR XCHECK
FMS200	FMS ERROR NUMBER fms-error-nbr fms-error-message PROJECT=project-name LIBRARY=library-name SECTION=section-name UNIT=unit-name

Msg. Key	Message Text
INF014	END OF INPUT CARDS
INF099	LIBRARY=library-name SECTION=section-name WRITTEN TO BACKUP-FILE
INF101	LIBRARY=library-name SECTION=section-name TERMINATED

MSG 76	UNIT higher-unit-name LINE line-nbr
	ERROR WHILE READING UNIT
	PROJECT=project-name LIBRARY=library-name
	SECTION=section-name UNIT=unit-in-error

MSG 77	UNIT higher-unit-name LINE line-nbr UNIT NOT FOUND IN LIBRARIES UNIT=unit-in-error

MSG 78	UNIT higher-unit-name LINE line-nbr
	INVALID UNIT TYPE FOR INCLUDE
	PROJECT=project-name LIBRARY=library-name
	SECTION=section-name UNIT=unit-in-error

MSG 79	UNIT higher-unit-name
	INCLUDED UNITS ARE NESTED IN RECURSIVE LOOP.
	PROJECT=project-name LIBRARY=library-name
	SECTION=section-name UNIT=unit-in-error

MSG 80	UNIT higher-unit-name LINE line-nbr INCLUDED UNITS ARE NESTED BEYOND LIMIT OF 50 LEVELS PROJECT=project-name LIBRARY=library-name SECTION=section-name UNIT=unit-in-error
MSG 83	UNIT link-unit-name REQUESTED LINK UNIT NOT FOUND
MSG 84	UNIT object-unit-name ERROR IN FORMAT OF OBJECT MODULE
MSG 85	UNIT object-unit-name SKELETON FOR OBJECT STUB NOT FOUND
MSG 111	UNIT unit-name LINE line-nbr CASE ENTRY STATEMENT MISSING NUMERIC IDENTIFIER. ERRONEOUS CODE MAY RESULT.

Msg. Key	Message Text
MSG 112	unit unit-name LINE line-nbr DO STATEMENT MISSING WHILE OR UNTIL. WHILE ASSUMED.
MSG 113	UNIT unit-name LINE line-nbr CASE FIGURE HAS NO CASE.
MSG 114	UNIT unit-name LINE line-nbr EXTRA ELSECASE FOUND FOR CASE FIGURE DISCARDED.
MSG 115	UNIT unit-name LINE line-nbr CASE STATEMENT FOUND AFTER ELSECASE WILL NOT BE EXECUTABLE.
MSG 116	UNIT unit-name LINE line-nbr EXTRA ELSE STATEMENT FOR IF FIGURE DISCARDED.
MSG 117	UNIT unit-name LINE line-nbr new-figure ENCOUNTERED BEFORE IF FIGURE TERMINATED. ENDIF ASSUMED PRECEDING new-figure.
MSG 118	UNIT unit-name LINE line-nbr new-figure ENCOUNTERED BEFORE DO FIGURE TERMINATED. ENDDO ASSUMED PRECEDING new-figure.
MSG 119	new-figure ENCOUNTERED BEFORE CASE FIGURE TERMINATED. ENDCASE ASSUMED PRECEDING new-figure.
MSG 120	UNIT unit-name LINE line-nbr UNMATCHED new-figure DELETED.
MSG 121	UNIT unit-name LINE line-nbr EXPECTED CASE NUMBER. FOUND WORD user-word. WORD DISCARDED.
MSG 122	UNIT unit-name LINE line-nbr OPTION CARD input-card-image IN ERROR. DEFAULT OPTIONS - NOSOURCE, MAP - ASSUMED.
MSG 123	UNIT unit-name LINE line-nbr NEST-STACK OVERFLOWED. MAXIMUM NUMBER OF NESTED STRUCTURED FIGURES 50 EXCEEDED. FATAL ERROR. EXECUTION TERMINATED.
MSG 124	UNIT unit-name LINE line-nbr CONDITION-STACK OVERFLOWED. NESTED DO CONDITIONS OCCUPY MORE THAN MAXIMUM 50 LINES. FATAL ERROR. EXECUTION TERMINATED.

Msg. Key	Message Text
MSG 125	UNIT unit-name LINE line-nbr CASE-STACK OVERFLOWED. MAXIMUM NUMBER OF NESTED CASES 10 EXCEEDED. FATAL ERROR. EXECUTION TERMINATED.
MSG 126	UNIT unit-name LINE line-nbr CASE-LABEL-STACK OVERFLOWED. MAXIMUM NUMBER OF CASE NUMBERS 200 EXCEEDED. FATAL ERROR. EXECUTION TERMINATED.
MSG 127	UNIT unit-name LINE line-nbr LABEL-STACK OVERFLOWED. MAXIMUM LABELS 125 EXCEEDED. CAUSED BY TOO MANY NESTED STRUCTURE FIGURES. FATAL ERROR. EXECUTION TERMINATED.
MSG 128	UNIT unit-name LINE line-nbr OUTPUT-STACK OVERFLOWED. PROBABLY CAUSED BY TOO MANY BLANK OR CONTINUATION LINES FOR ONE STATEMENT. FATAL ERROR. EXECUTION TERMINATED.
MSG 129	UNIT unit-name LINE line-nbr PROCEDURE DIVISION NOT FOUND. FATAL ERROR. EXECUTION TERMINATED.
MSG 130	UNIT unit-name LINE line-nbr MAXIMUM NUMBER OF STRUCTURE FIGURES 9999 EXCEEDED. FATAL ERROR. EXECUTION TERMINATED.
MSG 132	UNIT unit-name LINE line-nbr ERROR IN FORMAT OF INCLUDE CARD IN LINK UNIT input-card-image
MSG 163	UNIT unit-name LINE line-nbr NON-COMMENT WITH PUNCH IN COLUMN 6 WITHOUT A PRECEEDING STATEMENT CARD
MSG 164	UNIT unit-name LINE line-nbr END IF before IF, IF THEN, on ELSE encountered.
MSG 165	UNIT unit-name LINE line-nbr END WHILE ON ENDDO before DO WHILE encountered
MSG 166	UNIT unit-name LINE line-nbr END CASE before CASE OF, CASENTRY, CASE, CASE ELSE, or ELSE CASE encountered

Msg. Key	Message Text
MSG 167	UNIT unit-name LINE line-nbr END UNTIL or ENDDO before DO UNTIL encountered
MSG 168	UNIT unit-name LINE line-nbr ELSE after ELSE
MSG 169	UNIT unit-name LINE line-nbr CASE after CASE ELSE or ELSE CASE
MSG 170	UNIT unit-name LINE line-nbr CASE ELSE after CASE ELSE or ELSE CASE: or ELSE CASE after CASE ELSE or ELSE CASE
MSG 171	UNIT unit-name LINE line-nbr ELSE before IF encountered
MSG 172	UNIT unit-name LINE line-nbr CASE before CASE OF or CASENTRY encountered
MSG 173	UNIT unit-name LINE line-nbr Structural/syntactic error in program, cause unknown
MSG 174	UNIT unit-name LINE line-nbr CASE ELSE before CASE OF, CASENTRY, or CASE encountered
MSG 175	UNIT unit-name LINE line-nbr ENDDO statement encountered not preceeded by a DO WHILE or DO UNTIL statement
MSG 176	UNIT unit-name LINE line-nbr Character string too long in CASE statement
MSG 177	UNIT unit-name LINE line-nbr Program END without balancing blocks for each statement
MSG 178	UNIT unit-name LINE line-nbr Improperly nested DMATRAN statements
MSG 179	UNIT unit-name LINE line-nbr Cause unknown. Unrecognizable statement.
MSG 180	UNIT unit-name LINE line-nbr INCLUDE statement error encountered

Msg. Key	Message Text
MSG 181	UNIT unit-name LINE line-nbr ERROR encountered when attempting to retrieve a stub unit.
PSL012	BLOCK NUMBER TO BE RELEASED IS NOT IN RANGE UNDER LIBRARY=library-name SECTION=section-name BLOCK block-nbr FILE file-nbr

PSL021	BLOCK NUMBER IN ERROR FOR WRITE ON SECTION FILE
	BLOCK block-nbr FILE file-nbr
	PROJECT=project-name LIBRARY=library-name
	SECTION=section-name UNIT=unit-name

PSL022	INVALID FILE NUMBER file-nbr
PSL024	UNABLE TO ADD ENTRY TO INDEX FOR UNIT unit-name

PSL027	UNABLE TO LOCATE DIRECTORY FOR
	LIBRARY=library-name
	SECTION=section-name
	BLOCK block-nbr FILE file-nbr

PSL029	ERROR WHILE CHANGING INDEX ENTRY
PSL031	ERROR WHILE FINDING INDEX ENTRY
PSL033	UNABLE TO READ BLOCK block-nbr FILE file-nbr
PSL034	UNABLE TO WRITE BLOCK block-nbr FILE file-nbr

PSL037	UNABLE TO RELEASE FILE WITH
	ACCESS=access-mode FILE=file-nbr
	PROJECT=project-name LIBRARY=library-name
	SECTION=section-name UNIT=unit-name

PSL038	ILLEGAL PROCESS TYPE FOR INCLUDE. PROCESS TYPE MUST BE ADD OR DELETE.
PSL039	UNABLE TO DELETE INCLUDE FOR UNIT=unit-name
PSL040	UNABLE TO ADD INCLUDE FOR UNIT=unit-name
PSL041	UNABLE TO ASSIGN BLOCK IN LIBRARY=library-name SECTION=section-name

Msg. Key	Message Text
PSL042	UNABLE TO RELEASE BLOCK IN LIBRARY=library-name SECTION=section-name
PSL043	UNABLE TO INITIALIZE WRITE FOR UNIT=unit-name
PSL044	UNABLE TO WRITE LINE FOR UNIT=unit-name
PSL045	UNABLE TO TERMINATE WRITE FOR UNIT=unit-name
PSL046	UNABLE TO WRITE ACCOUNTING INFORMATION FOR UNIT=unit-name
PSL047	UNABLE TO INITIALIZE READ FOR UNIT=unit-name
PSL048	UNABLE TO READ LINE FOR UNIT=unit-name
PSL049	UNABLE TO TERMINATE READ FOR UNIT=unit-name
PSL051	UNABLE TO READ INDEX FILE file-nbr BLOCK block-nbr DIRECTORY yes-or-no
PSL052	UNABLE TO SPAWN JOB TO COMPLETE PROCESSING
PSL073	NESTING OF READS EXCEEDS (max-nest-level) LEVELS
PSL074	INVALID ACCESS=access-mode FOR UNIT TYPE=unit-type
PSL089	ILLEGAL PSL CARD PROCESSING AFTER END OF PSL INPUT
PSL094	INVALID BLOCK NUMBER FOR RANDOM FILE ACCESS BLOCK block-nbr FILE file-nbr
PSL096	UNABLE TO GENERATE STUB FOR INCLUDED UNIT. DID NOT DELETE UNIT=unit-name
PSL139	RECORD-SEQUENCE-ERROR ENCOUNTERED ON BACKUP TAPE AT RECORD-SEQUENCE-NBR=seq-number ON RECORD-TYPE-NBR=type-number

APPENDIX D. PSL PROCEDURES

The following pages of Appendix D contain the JCL procedures which are provided with the PSL system to invoke the PSL programs. The procedures, which are stored as units in the JOB section, are as follows:

a. ASM - Assembles ASM code.

b. RUN - Invokes the Batch PSL system.

c. CLMD - Collects management data.

d. COBOL - Compiles COBOL code without using a precompiler.

e. FORTRAN - Compiles FORTRAN code without using a precompiler.

f. MD - Prints Management Data report.

g. PS - Print Program Structure report.

h. SCOBOL - Processes COBOL code with precompiler and compiles resulting code.

i. SPFORT - Processes COBOL code with recompiler and compiles resulting code.

j. COBOLG - Processes unstructured COBOL using the General Preprocessor and compiles resulting code.*

k. FORTRANG - Processes unstructured FORTRAN using the General Preprocessor and compiles resulting code.*

l. PSLCOB - Processes COBOL code with precompiler COPYLIB and compiles resulting code.

m. ASMG - Processes unstructured ASM using the General
Preprocessor
and compiles resulting code.*

Each of the procedures, except RUN, is used in a spawned job and is modified by the PSL system to reference appropriate file names before the procedure is spawned.

*See Appendix I.

FIGURE D-01 PROCEDURE ASM

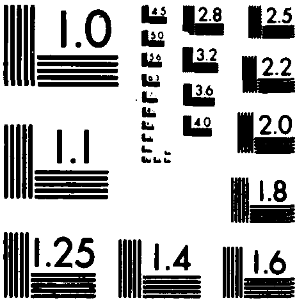
```
@ASG,CP MS,F///100
@ASG,A MS,F///100
@ASG,T JS,F///100
@ASG,T UC,F///100
@ASG,CP UL,F///100
@ASG,A UL,F///100
@ASG,T FC,F///100
@ASG,T PK,F///100
@ASG,T RC,F///100
@ASG,T NC,F///100
@ASG,T UT,F///100
@ASG,T FJ,F///100
@ASG,T FS,F///100
@ASG,T TF,F///100
@ASG,T CHF,F///10
@ASG,T XF,F///100
@ASG,T XL,F///100
@ASG,T NF,F///100
@ASG,T CF,F///100
@ASG,T FT,F///100
@ASG,T YA,F///100
@ASG,T F1,F///100
@ASG,T F2,F///100
@ASG,T F3,F///100
@ASG,T R0,F///100
@ASG,T R1,F///100
@ASG,T R2,F///100
@ASG,T R3,F///100
@ASG,T R4,F///100
@DATA,I CC.,,PSL
@END PSL
@XQT PSL.BCTL
```

FIGURE D-02 PROCEDURE RUN

AD-A107 249 IBM FEDERAL SYSTEMS DIV GAITHERSBURG MD F/G 9/2
PROGRAMMING SUPPORT LIBRARY (PSL). USERS MANUAL.(U)
MAY 78 F30602-77-C-0249
UNCLASSIFIED DOD/DF-81/016A NL

3 OF 3

END
DATE
FILMED
12-81
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
@ASG,CP MS.
@ASG,A MS.
@DATA,I CC.,INPUT
@END INPUT
@XQT PCL.CLMD
@FREE MS
@SYM MS.,PR
```

INPUT

FIGURE D-03 PROCEDURE CLMD

```
1 @ACOB,TSEC                                                            COMPILE
2 @ADD                                                                  SOURCE
3 @EOF
```

FIGURE D-04 PROCEDURE COBOL

```
1 @FTN,SWT                                                                COMPILE
2 @ADD                                                                    SOURCE
3 @EOF
```

FIGURE D-05 PROCEDURE FORTRAN

INPUT

```
@ASG,CP MS.
@ASG,A MS.
@ASG,CP LS.
@ASG,A LS.
@DATA,I CC,,INPUT
@END INPUT
@XQT PSL.PRMD
@FREE MS
@FREE LS
@SYM MS,,PR
@SYM LS,,PR
```

FIGURE D-06 PROCEDURE MD

```
@ASG,CP PR.
@ASG,A PR.
@ASG,CP LS.
@ASG,A LS.
@DATA,I CC,,INPUT
@END INPUT
@XQT PSL.PRPS
@FREE PR
@FREE LS
@SYM PR.,,PR
@SYM LS.,,PR
```

INPUT

FIGURE D-07 PROCEDURE PS

```
@ASG,CP PR.
@ASG,A PR.
@ASG,CP LS.
@ASG,A LS.
@ASG,T PC.
@DATA,I CC,,INFLT                                                  INPUT
@END INPUT
@XQT FSL.PPCP
@FREE PF
@FREE LS
@SYM PF,,PR
@SYM LS,,PR
@ACOB,TSEC                                                         COMPILE
@ADD PC. .
@EOF
```

FIGURE D-08 PROCEDURE SCOBOL

```
@ASG.CP PR.
@ASG,A PR.
@ASG,CP LS.
@ASG.A LS.
@ASG.T FO.
@DATA,I CC,,INPUT                                      INPUT
@END INPUT
@XQT PSL.PRGL
@ASG,T 4,F///100
@DATA,I 4.,,FORTIN
@ADD,D FO. .
@END FORTIN
@FREE PR
@FREE LS
@SYM PR,,PR
@SYM LS,,PR
@FREE 2.
@ASG,T 2,F///100
@USE NEW.,2.
@XQT PSL.PRFT
@FOR,SWT                                               COMPILE
@ADD NEW. .
@EOF
```

FIGURE D-09 PROCEDURE SPFORT

```
@ASG,CP PR.
@ASG,A FR.
@ASG,CP LS.
@ASG,A LS.
@ASG,T PC.
@DATA,I CC,,INPUT                                                    INPUT
@END INPUT
@XQT PSL.PPCL
@FREE FR
@FREE LS
@SYM PR,,FR
@SYM LS,,PR
@ACOB,TSEC                                                           COMPILE
@ADD PC. .
@EOF
```

FIGURE D-10 PROCEDURE COBOLG

```
@ASG,CF PR.
@ASG,A PR.
@ASG,CF LS.
@ASG,A LS.
@ASG,T PC.
@DATA,I CC,,INPUT                                                    INPUT
@END INPUT
@XQT FSL.PPCL
@FREE PR
@FREE LS
@SYM PR,,PR
@SYM LS,,FR
@FOR,SWT                                                             COMPILE
@ADD PC. .
@EOF
```

FIGURE D-11 PROCEDURE FORTRAN

```
@ASG,CP PR.
@ASG,A PR.
@ASG,CP LS.
@ASG,A LS.
@ASG,T PC.
@DATA,I CC,,INPUT                                        INPUT
@END INPUT
@XQT PSL.FPCL
@FREE PR
@FREE LS
@SYM PR,,PF
@SYM LS,,PR
@ASM,WL                                                  COMPILE
@ADD PC. .
@EOF
```

FIGURE D-12 PROCEDURE ASMG

```
@USE COBSPF.,S.
@ASG,CP PP.
@ASG,A PR.
@ASG,CP LS.
@ASG,A LS.
@ASG,T FC.                                                          INPUT
@DATA,I CC.,INPUT
@END INPUT
@XQT PSL.PROC
@FREE PR
@FREE LS
@SYM PR,,PP
@SYM LS,,PP                                                         COMPILE
@COB,TSEC
@ADD PC. .
@EOF
```

FIGURE D-13 PROCEDURE PSLCOB

APPENDIX E. FILE NAMES

The following file names are used by the PSL system:

a. Regular Processing (not spawned)

1.	CC	Input of PSL Function cards and data
2.	CHF	Temporary work file
3.	F1-F3	Independent unit files (sequential)
4.	FJ	JCL for spawned job
5.	FS	Temporary work file
6.	JS	Temporary work file
7.	MS	Listing of input cards and messages
8.	R1-R4	PSL Standard files (random)
9.	R0	PSL Standard file (random)
10.	RT	Input to RESTORE
11.	TF	Temporary work file
12.	UC	Punched card output (SOURCE,MEDIA=CARD)
13.	UL	Listings of units and indexes
14.	UT	BACKUP output or output from (SOURCE,MEDIA=TAPE)
15.	XA	Sort work file
16.	XF	Work file

b. Spawned Jobs

1.	02	Generated source code for input to FORTRAN compiler
2.	CC	Input of PSL Function cards and data
3.	JV	Output from COMPOOL assembly
4.	J0-J9	COMPOOL input files (symbol tables)
5.	F1-F3	Independent unit files
6.	LS	Program structure report
7.	MS	Message listing
8.	NC	New collection - work file
9.	PK	Plan keywords - work file
10.	PO	Generated source code for input to compiler
11.	PR	Message output
12.	R1-R9	PSL Standard files (random)
13.	RC	Random collection - work file
14.	XA	Sort work file
15.	XF	Work file

APPENDIX F. USERS MANUAL FOR PSL STRUCTURED FORTRAN PRECOMPILER

The following pages of this appendix provide the relevant information on the FORTRAN precompiler of the PSL. This precompiler was developed under the name of STRUCTRAN-1* by General Research Corporation for RADC, and has been installed in the PSL by IBM.

F.1 Introduction

The techniques of Structured Programming are finding increasing application in the computing community. Structured programs are, however, difficult to write in programming languages that do not have the statement types necessary to produce GOTO-free code. SPFORT is a programming language that augments standard FORTRAN to permit its use as a basis for structured programming. Six structured programming statement forms are provided for use in FORTRAN-based software development. The FORTRAN preprocessor translates SPFORT into standard FORTRAN. It accepts as input modules containing both SPFORT and FORTRAN statements, and produces as output indented pure-FORTRAN modules logically equivalent to the input modules. Features include error processing to warn of improper statement forms, and warnings to indicate the presence of unstructured FORTRAN control statements.

The structured-programming statement forms implemented in the FORTRAN precompiler are described in Section F.2, with typical examples of SPFORT subroutines and the translated FORTRAN subroutines. Error diagnostics are described in Section F.3. SPFORT guidelines are summarized in Section F.4.

F.2 SPFORT Constructs

SPFORT replaces FORTRAN control statements with the following control statement constructs:

a. IF...THEN...ELSE...END IF--provides block structuring of conditionally executable sequences of statements.

b. DO WHILE...END WHILE--permits iteration of a code segment while a specified condition remains true.

*For additional information regarding this capability as it was developed, see the Final Technical Report, Structured Programming Translators, RADC-TR-76-253, Volumes I to V.

c. CASE OF...CASE...CASE ELSE...END CASE--allows multiple choices for selecting program action.

d. DO UNTIL...END UNTIL--permits iteration until a specified condition becomes true.

e. BLOCK NAME...END BLOCKS (and corresponding INVOKE NAME statement)--provide internal subroutines.

f. INCLUDE--provides for top-down programming and implementation.

These statement forms can be intermixed with standard FORTRAN non-control statements in the text stream which is processed by the FORTRAN preprocessor. SPFORT statements are converted by the preprocessor to their FORTRAN equivalents, and the resulting file can be compiled by the FORTRAN compiler in the normal manner.

The following simple examples illustrate the use of these control statements.

F.2.1 IF...THEN...ELSE...END IF

The general form of the IF construct consists of three SPFORT statements:

```
IF(<logical-expression>)[THEN]

   <block of statements>

[ELSE                        ]
[                            ]
[  <block of statements>     ]
[                            ]
ENDIF
```

where<logical-expression>is any legal FORTRAN logical expression. The word TEHN is optional. The<logical-expression>is evaluated, and if it is true, the statements following the IF are executed until the ELSE is reached, where control passes to the first statement after the END IF. The ELSE is optional. If the ELSE is absent and the<logical-expression>is false, control passes to the first statement after the END IF. If the ELSE is present and the<logical-expression>is false, control passes to the statement following the ELSE so that the statements after the ELSE are executed. An example SPFORT subroutine containing the IF construct is presented in Figure F-01a. An alternate format for writing the IF construct is shown in Figure F-01b. The STRUCTRAN-1 translation of the example subroutine is presented in Figure F-01c.

F.2.2 DO WHILE...END WHILE

The general form of the DO WHILE construct consists of two SPFORT statements:

```
DO WHILE(< logical-expression >)

  < block of statements >

END WHILE
```

where < logical-expression > is any legal FORTRAN logical expression. The DO WHILE construct represents an iteration in which execution occurs in the following manner:

a. The value of logical-expression is found: if true, the statements contained within the DO WHILE block are executed; if false, control passes to the statement immediately following the END WHILE.

b. If the statements within the DO WHILE block have been executed, the value of logical-expression is checked again, with the same consequences as in a.

The iterative block in the DO WHILE...END WHILE may be executed zero or more times. In general, it is necessary to initialize the loop control variable before entering the DO WHILE construct, and also necessary to modify it within the DO WHILE construct. An example SPFORT subroutine containing the DO WHILE construct is presented in Figure F-02a. An alternate format for writing this construct is shown in Figure F-02b. The FORTRAN preprocessor generates the translation of the example subroutine presented in Figure F-02c.

The IF construct and the DO WHILE construct are sufficient to express the control portion of any algorithm which can be implemented in FORTRAN. However, for greatest convenience in implementation of software systems with structured programming techniques, some additional statement forms are highly desirable. The CASE construct is a selection structure and the DO UNTIL is an iteration structure, while the BLOCK construct enhances modularity.

```
(a)  SUBROUTINE IFTHEN (IN,OUT)
     INTEGER OUT
     IF (IN.GE.10) THEN
        OUT=0
     ELSE
        OUT=OUT+1
     END IF
     RETURN
     END
```

```
(b)  SUBROUTINE IFTHEN (IN,OUT)
     INTEGER OUT
     IF (IN.GE.10)
        OUT=0
     ELSE
        OUT=OUT+1
     ENDIF
     RETURN
     END
```

```
(c)        SUBROUTINE IFTHEN (IN,OUT)
           INTEGER OUT
           IF (IN.GE.10) GO TO 99998
           GO TO 99997
     99998 CONTINUE
              OUT=0
           GO TO 99996
     99997 CONTINUE
              OUT=OUT+1
     99996 CONTINUE
           RETURN
           END
```

Figure F-01. SPFORT IF...THEN...ELSE...ENDIF Construct With FORTRAN Preprocessor Translation

```
(a)   SUBROUTINE DOWHILE (IN,OUT,I)
      INTEGER OUT
      DIMENSION IN(50)
      I=1
      DO WHILE (IN(I).NE.OUT.AND.I.LE.50)
         I=I+1
      END WHILE
      RETURN
      END
```

```
(b)   SUBROUTINE DOWHIL (IN,OUT,I)
      INTEGER OUT
      DIMENSION IN(50)
      I=1
      DOWHILE (IN(I).NE.OUT.AND.I.LE.50)
         I=I+1
      ENDDO
      RETURN
      END
```

```
(c)         SUBROUTINE DOWHILE (IN,OUT,I)
            INTEGER OUT
            DIMENSION IN(50)
            I=1
      99998 IF (IN(I).NE.OUT.AND.I.LE.50) GO TO 99997
            GO TO 99996
      99997 CONTINUE
              I=I+1
            GO TO 99998
      99996 CONTINUE
            RETURN
            END
```

Figure F-02. SPFORT DO WHILE...END WHILE Constructs With FORTRAN Preprocessor Translation

F.2.3 CASE OF...CASE...CASE ELSE...END CASE

The CASE statement provides a way to select which group of statements will be executed. The general form of the CASE construct consists of the following SPFORT statements:

```
     CASE OF(< integer-expression >)

     CASE(< i >)

     CASE(< j >)

        < block of statements >

     CASE ELSE

        < block of statements >

     END CASE
```

< i > and < j > represent integers of positive value. They may be in any order, and there is no limit to how many integers may be listed. In a list of integers appearing on the same CASE statement, commas are required between integers.

The < integer-expression > is computed, and if any of the specified integers in the CASE list are equal to the value of the expression, then the transfer of control is to the statements which follow the particular CASE. If there is no such CASE, and the CASE ELSE statement is present, then the block of statements following the CASE ELSE is executed; otherwise, no block is executed. If there are two CASE statements with the same CASE index, the first occurring one is executed (if the CASE expression has that value). After the block of statements selected has been executed, control transfers to the statement after the END CASE.

An example SPFORT subroutine containing the CASE construct is presented in Figure F-03a. An alternate form of writing this construct is shown in Figure F-03b. The FORTRAN preprocessor produces the translation of the example subroutine shown in Figure F-03c.

F.2.4 DO UNTIL...END UNTIL

The general form of the DO UNTIL construct consists of two SPFORT statements:

```
(a)  SUBROUTINE CASE (IN,OUT)
     INTEGER OUT
     CASE OF (IN)
     CASE (10)
        OUT=IN
     CASE (15)
        OUT=IN-3
     CASE ELSE
        OUT=IN+10
     END CASE
     RETURN
     END
```

```
(b)  SUBROUTINE CASE (IN,OUT)
     INTEGER OUT
     CASENTRY (IN)
     CASE  (10)
       OUT=IN
     CASE  (15)
       OUT=IN-3
     ELSECASE
       OUT=IN+10
     ENDCASE
     RETURN
     END
```

```
(c)         SUBROUTINE CASE (IN,OUT)
            INTEGER OUT
            I99998=IN
            IF (I99998.NE.(10)) GO TO 99996
               OUT=IN
            GO TO 99997
      99996 CONTINUE
            IF (I99998.NE.(15)) GO TO 99995
               OUT=IN-3
            GO TO 99997
      99995 CONTINUE
               OUT=IN+10
      99997 CONTINUE
            RETURN
             END
```

Figure F-03. SPFORT CASE Construct With FORTRAN Preprocessor Translation

```
      DO UNTIL(< logical-expression >)

         < block of statements >

      END UNTIL
```

The statements enclosed within the DO UNTIL and the END UNTIL are always executed once. Then the < logical-expression > is evaluated and, if false, iteration and evaluation of the expression continue until it is true. At that time execution of the statements following the END UNTIL begins.

An example SPFORT subroutine containing the DO UNTIL construct is presented in Figure F-04a. An alternate format for writing this construct is presented in Figure F-04b. The FORTRAN preprocessor produces the translation of this subroutine shown in Figure F-04c. After completion of the DO UNTIL iteration, < J > will have the value 16 and < I > the value 11. It is important to note that when using DO WHILE or DO UNTIL constructs, the iteration variable must be initialized before entering the iteration and modified within the iteration.

F.2.5 BLOCK...END BLOCK and INVOKE

The constructs described in the preceding paragraphs allow most programming tasks to be done in a well-structured manner. However, they do not always permit top-down programming. To implement this method, one must be able to refer to an action (such as "compute array element") before the code for it is actually available.

The usual method for doing this is calling subroutines. However, subroutines have certain disadvantages. The overhead involved in calling them is often high. Additionally, those variables used in both a calling routine and a called subroutine must either be placed in COMMON or passed as parameters.

In many cases, a subroutine uses only variables which are already in the routine which calls it. Use of a subroutine internal to the calling routine eliminates the need for any mechanism (such as parameters or COMMON blocks) for referring to the variables required.

A facility for creating and using this type of subroutine has been added to SPFORT. This construct is called a BLOCK which may be defined as an internal parameterless procedure with all variables global. A BLOCK can be called only from the individual

```
(a)  SUBROUTINE DOUNTL (IN,OUT)
     DIMENSION IN(10)
     INTEGER OUT(20)
     I=1
     J=6
     DO UNTIL (I.GT.10)
        OUT(J)=IN(I)
        I=I+1
        J=J+1
     END UNTIL
     RETURN
     END
```

```
(b)  SUBROUTINE DOUNTL (IN,OUT)
     DIMENSION IN(10)
     INTEGER OUT(20)
     I=1
     J=6
     DOUNTIL (I.GT.10)
        OUT(J)=IN(I)
        I=I+1
        J=J+1
     ENDDO
     RETURN
     END
```

```
(c)         SUBROUTINE DOUNTL (IN,OUT)
            DIMENSION IN(10)
            INTEGER OUT(20)
            I=1
            J=6
            GO TO 99998
      99997 IF (I.GT.10) GO TO 99996
      99998 CONTINUE
               OUT(J)=IN(I)
               I=I+1
               J=J+1
            GO TO 99997
      99996 CONTINUE
            RETURN
            END
```

Figure F-04. SPFORT DO UNTIL...END UNTIL Construct With FORTRAN Preprocessor Translation routine (main program, subroutine, or function) in which it is compiled; it cannot be called from an external routine, nor can it be passed as a parameter to another routine. A BLOCK is simply a segment of the code of the routine which contains it. The BLOCK is exercised only if it is invoked.

The general form of a BLOCK construct consists of two SPFORT statements:

```
     BLOCK(< block-name >)

        < statements >

     END BLOCK
```

where < block-name > is any string of characters (e.g., COMPUTE, INDEX, or PRINT-CURRENT-STATUS). The name of a BLOCK may be arbitrarily long, so that the name can have mnemonic significance. However, the first six characters must be unique.

A BLOCK is called by an INVOKE statement, whose format is:

```
     INVOKE(< block-name >)
```

When an INVOKE statement is executed, control is transferred to the first statement in the BLOCK; when the END BLOCK is reached, control goes to the statement following the INVOKE of the BLOCK. Though BLOCKs can be nested (one BLOCK completely inside of another), no recursion is allowed in the calling of BLOCKs (i.e., a BLOCK cannot invoke itself). Also, the name of a BLOCK is known throughout the entire routine in which it is contained.

The following are examples of the two major uses of the BLOCK construct:

Example 1: Top-Down Programming

```
I=1
DO UNTIL (I .GT. N)
    J=1
    DO UNTIL (J .GT. N)
       INVOKE(COMPUTE.ARRAY.ELEMENT)
       J=J+1
    END UNTIL
    I=I+1
END UNTIL
```

and, at some place later in the same routine:

```
BLOCK(COMPUTE.ARRAY.ELEMENT)
    code to compute A(I,J)
    A(I,J) = value computed
    J=J+1
END BLOCK
```

The use of a BLOCK construct enhances readability and understandability of the program. When a block of code is invoked only once, the INCLUDE capability can be conveniently used in place of the BLOCK construct.

Example 2: Internal Subroutine

S_1 and S_2 in the following code represent two sets of statements. The use of a BLOCK in Method 2 below eliminates the need for duplicating code.

```
Method 1:  IF(A) THEN
              IF(C) THEN
                 $S_1$
              ELSE
                 $S_2$
              ENDIF
           ELSE
              IF(D) THEN
                 $S_2$
              ELSE
                 $S_1$
              ENDIF
           ENDIF
```

```
Method 2:  IF(A) THEN
              IF(C) THEN
                 INVOKE(BLOCK-A)
              ELSE
                 INVOKE(BLOCK-B)
              END IF
           ELSE
              IF(D) THEN
                 INVOKE(BLOCK-B)
              ELSE
                 INVOKE(BLOCK-A)
              END IF
           END IF
```

where the BLOCKs are defined as:

```
BLOCK(BLOCK-A)
    $S_1$
END BLOCK
```

and

```
BLOCK(BLOCK-B)
    $S_2$
END BLOCK
```

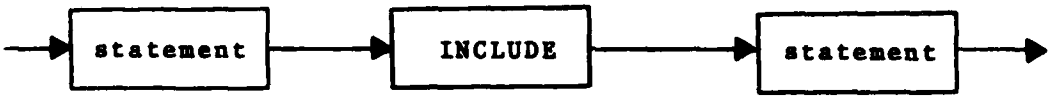
There is a maximum of 20 BLOCKs per module with a limit of 15 INVOKEs for any one BLOCK. The overhead in time and space involved in using BLOCKs is less than that of using subroutine calls.

An example SPFORT subroutine containing the INVOKE...BLOCK CONSTRUCT is presented in Figure F-05a. The FORTRAN preprocessor produces the translation of this construct presented in Figure F-05b.

F.2.6 INCLUDE

An INCLUDE statement is used to refer to a unit which will be retrieved and substituted in-line for the INCLUDE statement.

The flowchart for the INCLUDE figure is:



and the code structure to be used to represent the INCLUDE is:

	Format	
statement		
numerics	INCLUDE	unit-name
statement		

A more detailed description of INCLUDE statements is contained in Section 3.2.8.

F.3 Using the FORTRAN Preprocessor

Figure F-06 illustrates an SPFORT source program ready for input to the FORTRAN preprocessor. The SPFORT source code has not been manually indented in order to avoid the problem of updating indentation levels as the program is modified. More than one module may be processed in each preprocessor run.

```
(a)   SUBROUTINE BLOCK (WIDTH,LENGTH)
      INTEGER AREA,WIDTH
      LENGTH=LENGTH+20
      WIDTH=WIDTH+30
      INVOKE (COMPUTE AREA)
      INVOKE (PRINT AREA)
      BLOCK (COMPUTE AREA)
         AREA=LENGTH*WIDTH
      END BLOCK
      BLOCK (PRINT AREA)
         WRITE (6,1)AREA
    1    FORMAT (10X,I20)
      END BLOCK
      RETURN
      END
```

```
(b)          SUBROUTINE BLOK (WIDTH,LENGTH)
             INTEGER AREA,WIDTH
             LENGTH=LENGTH+20
             WIDTH=WIDTH+30
             ASSIGN 99997 TO I99998
             GO TO 99998
       99997 CONTINUE
             ASSIGN 99995 TO I99996
             GO TO 99996
       99995 CONTINUE
             GO TO 99994
       99998 CONTINUE
               AREA=LENGTH*WIDTH
             GO TO I99998,(99997)
       99994 CONTINUE
             GO TO 99993
       99996 CONTINUE
               WRITE (6,1)AREA
           1   FORMAT (10X,I20)
             GO TO I99996,(99995)
       99993 CONTINUE
             RETURN
```

Figure F-05. SPFORT INVOKE...BLOCK Construct With FORTRAN Preprocessor Translation

```
      SUBROUTINE EXAMPL (INFO,LENGTH)                            EXAMPL1
C                                                                EXAMPL2
C     ILLUSTRATION OF SPFORT SYNTAX                              EXAMPL3
C                                                                EXAMPL4
      IF (INFO.LE.10 .AND. LENGTH.GT.0)THEN                      EXAMPL5
      INFO=INFO+10                                               EXAMPL6
      ELSE                                                       EXAMPL7
      LENGTH=50                                                  EXAMPL8
      END IF                                                     EXAMPL9
      CASE OF (INFO+6)                                           EXAMPL10
      CASE (14)                                                  EXAMPL11
      LENGTH=LENGTH-INFO                                         EXAMPL12
      CASE (17)                                                  EXAMPL13
      DO WHILE (INFO.LT.20)                                      EXAMPL14
      DO UNTIL (LENGTH.LE.INFO)                                  EXAMPL15
      INVOKE (COMPUTE LENGTH)                                    EXAMPL16
      IF (LENGTH.GE.30) THEN                                     EXAMPL17
      INVOKE (PRINT-RESULTS)                                     EXAMPL18
      END IF                                                     EXAMPL19
      END UNTIL                                                  EXAMPL20
      INFO=INFO+1                                                EXAMPL21
      END WHILE                                                  EXAMPL22
      CASE ELSE                                                  EXAMPL23
      DO WHILE (LENGTH.GT.0)                                     EXAMPL24
      INVOKE (COMPUTE LENGTH)                                    EXAMPL25
      END WHILE                                                  EXAMPL26
      END CASE                                                   EXAMPL27
      BLOCK (PRINT-RESULTS)                                      EXAMPL28
      WRITE (6,1)INFO,LENGTH                                     EXAMPL29
    1 FORMAT (10X,I5,20X,I5)                                     EXAMPL30
      END BLOCK                                                  EXAMPL31
      BLOCK (COMPUTE LENGTH)                                     EXAMPL32
      LENGTH=LENGTH-10                                           EXAMPL33
      END BLOCK                                                  EXAMPL34
      RETURN                                                     EXAMPL35
      END                                                        EXAMPL36
```

Figure F-06. FORTRAN Preprocessor Input

Figure F-07 illustrates the translated FORTRAN version of Figure F-06 produced by the preprocessor. This may be compiled and executed in the normal manner. The translated FORTRAN is indented to reflect the structure of the original SPFORT program. The sequence information in columns 73 through 80 refers back to the card number of the original SPFORT statement. This aids in relating FORTRAN error diagnostics to the SPFORT source code to be modified.

A list and description of the error messages are provided in Table F-1.

F.4 Guidelines

The following guidelines should be kept in mind when using the FORTRAN preprocessor:

a. A maximum of 20 cards per statement.

b. The FORTRAN preprocessor generates FORTRAN GOTO statements and statement labels. Statement labels in the SPFORT input source (FORMAT statements) should not duplicate the labels appearing in the translated FORTRAN.

c. When the DO UNTIL...END UNTIL construct is used for iteration, it is important to note that the statements contained within the construct will be executed once before the logical expression is evaluated.

d. All two word SPFORT directives may be written as two separate words or merged into one; e.g., DOUNTIL or DO UNTIL.

e. A maximum of 20 BLOCKS per module.

f. A limit of 15 INVOKEs for any one BLOCK.

g. INVOKE for the BLOCK must occur before the BLOCK code (suggest BLOCKs be grouped at the end of the routine).

h. First six characters of the name of a BLOCK must be unique within a module.

i. Maximum nexting level for indentation of FORTRAN output is 10.

```
       SUBROUTINE EXAMPL (INFO,LENGTH)                    1*
       IF (INFO.LE.10 .AND. LENGTH.GT.0) GO TO 99998      5*
       GO TO 99997                                        5*
99998  CONTINUE                                           5*
         INFO=INFO+10                                     6*
       GO TO 99996                                        7*
99997  CONTINUE                                           7*
         LENGTH=50                                        8*
99996  CONTINUE                                           9*
       I99995=INFO+6                                     10*
       IF (I99995.NE.(14)) GO TO 99993                   11*
         LENGTH=LENGTH-INFO                              12*
       GO TO 99994                                       13*
99993  CONTINUE                                          13*
       IF (I99995.NE.(17)) GO TO 99992                   13*
99991    IF (INFO.LT.20)  GO TO 99990                    14*
         GO TO 99989                                     14*
99990    CONTINUE                                        14*
           GO TO 99988                                   15*
99987      IF (LENGTH.LE.INFO) GO TO 99986               15*
99988      CONTINUE                                      15*
             ASSIGN 99983 TO I99984                      16*
             GO TO 99984                                 16*
99983        CONTINUE                                    16*
             IF (LENGTH.GE.30) GO TO 99982               17*
             GO TO 99981                                 17*
99982        CONTINUE                                    17*
               ASSIGN 99979 TO I99980                    18*
               GO TO 99980                               18*
99979          CONTINUE                                  18*
99981        CONTINUE                                    19*
           GO TO 99987                                   20*
99986      CONTINUE                                      20*
99985      CONTINUE                                      20*
           INFO=INFO+1                                   21*
         GO TO 99991                                     22*
99989    CONTINUE                                        22*
       GO TO 99994                                       23*
99992  CONTINUE                                          23*
99978    IF (LENGTH.GT.0) GO TO 99977                    24*
         GO TO 99976                                     24*
```

Figure F-07. Translated FORTRAN

```
99977    CONTINUE                                      24*
           ASSIGN 99975 TO I99984                      25*
           GO TO 99984                                 25*
99975      CONTINUE                                    25*
         GO TO 99978                                   26*
99976    CONTINUE                                      26*
99994  CONTINUE                                        27*
       GO TO 99974                                     28*
99980  CONTINUE                                        28*
         WRITE (6,1)INFO,LENGTH                        29*
    1    FORMAT (10X,I5,20X,I5)                        30*
       GO TO I99980,(99979)                            31*
99974  CONTINUE                                        31*
       GO TO 99973                                     32*
99984  CONTINUE                                        32*
         LENGTH=LENGTH-10                              33*
       GO TO I99984,(99983,99975)                      34*
99973  CONTINUE                                        34*
       RETURN                                          35*
       END                                             36*
```

Figure F-07. Translated FORTRAN (Continued)

j. The value of integer-expression in CASE statements must be positive.

k. BLOCK constructs cannot contain labeled statements which are referred to outside of the BLOCK.

l. A BLOCK cannot be entered by falling into it (as the next executable statement).

ERROR NUMBER	ERROR MESSAGE DESCRIPTION
163	Continuation card encountered without a preceding statement card.
164	END IF before IF, IF THEN, or ELSE encountered.
165	END WHILE or ENDDO before DO WHILE encountered.
166	END CASE before CASE OF, CASENTRY, CASE, CASE ELSE, or ELSE CASE encountered.
167	END UNTIL or ENDDO before DO UNTIL encountered.
168	ELSE after ELSE.
169	CASE after CASE ELSE or ELSE CASE.
170	CASE ELSE after CASE ELSE or ELSE CASE; or ELSE CASE after CASE ELSE or ELSE CASE.
171	ELSE before IF encountered.
172	CASE before CASE OF or CASENTRY encountered.
173	Structural/syntactic error in program, cause unknown*.
174	CASE ELSE before CASE OF, CASENTRY, or CASE encountered.
175	ENDDO statement encountered not preceded by a DO WHILE or DO UNTIL statement.

*This occurs when the indentation level computed by the preprocessor would become negative if the error was not detected. Possible causes are too many ELSE, END IF, END WHILE, END UNTIL, or END CASE statements.

Table F-1. Error Message Definitions

ERROR NUMBER	ERROR MESSAGE DESCRIPTION
176	Character string too long in CASE statement.
177	Program END without balancing blocks for each statement.
178	Improperly nested statements.
179	Cause unknown. Unrecognizable statements.
180	INCLUDE statement error encountered.
181	Error encountered when attempting to retrieve a stub unit.

Table F-1. Error Message Definitions (Continued)

APPENDIX G. USERS MANUAL FOR PSL STRUCTURED COBOL PRECOMPILER

G.1 Precompiler Objectives

The objective of the ANS COBOL precompiler is to assist the user in writing this language in a structured format. While the basic control logic figures required to accomplish this can be simulated in the COBOL language (refer to "Programming Language Standards", Volume I of the Structured Programming Series), the simulation of these figures is not as desirable as if the language itself permitted the implementation of such control logic directly. This is particularly true of the DO and CASE figures. When using the DO figure with the precompiler, the user may place the code block under control of the loop in-line rather than simulating it with a PERFORM statement that requires that it be placed out-of-line. Similarly, the simulated CASE figure requires that the programmer code explicit GO TO statements to the end of the figure, whereas with the precompiler syntax, such statements are generated automatically. Thus, the precompiler is intended to achieve the twin objectives of first, easing the task of writing structured COBOL, and second, improving the readability of such code thorugh the use of precompiler syntax rather than simulated structured COBOL.

G.2 Precompiler Inputs

The ANS COBOL precompiler accepts as input an ANS COBOL program written in precompiler syntax as defined in Section G.4. The structuring verbs described in that section are processed by algorithms described in Volume II of the Structured Programming Series to produce the output indicated in Section G.5. These structuring verbs should be viewed as complete sets which are designed to be processed as a unit. Three such sets are processed with this precompiler. They are:

a. the IF set

```
IF
ELSE (optional)
ENDIF
```

b. the DO set

```
DO
ENDDO
```

c. the CASE set

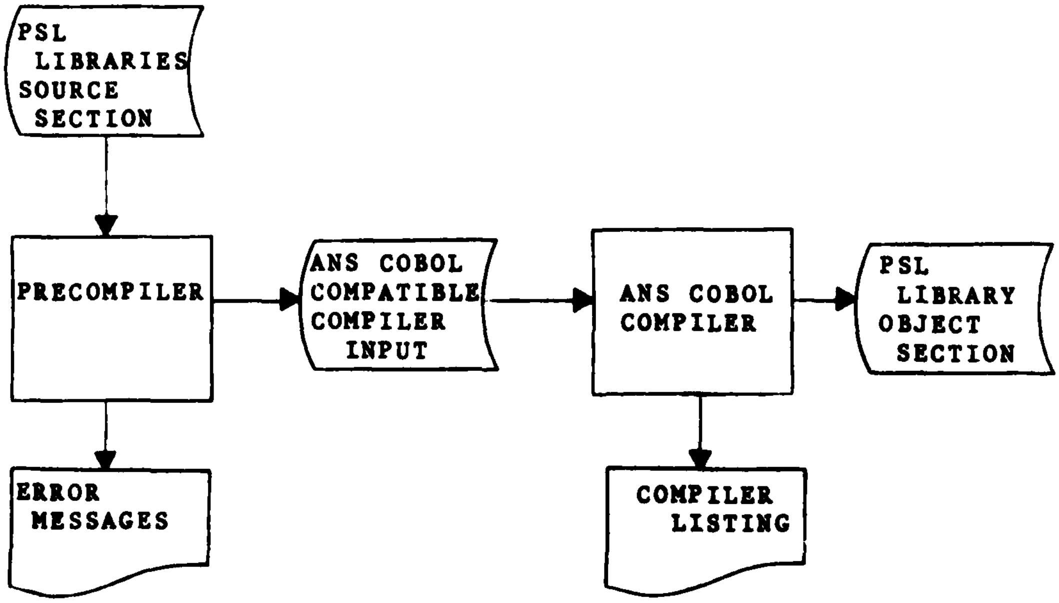
```
CASENTRY
CASE (at least one must be present)
ELSECASE (optional)
ENDCASE
```

G.3 Precompiler Output

The primary output of the precompiler is an ANS COBOL (X3.23-1968) compatible compiler input. This data set, input to the ANS COBOL compiler, is different from the compiler source code listing because of the intermediate translation. Comments are inserted to show the project and library where each unit of original source code was found. Original unit-line-numbers are placed in columns 1 through 6 of the precompiler output for programmer reference if necessary.

A second type of precompiler output consists of error messages as appropriate for the situation. Error messages are contained in Section G.7.

The relationship between the precompiler and its inputs and outputs is graphically displayed by the following:


PSL LIBRARIES SOURCE SECTION
PRECOMPILER
ANS COBOL COMPATIBLE COMPILER INPUT
ANS COBOL COMPILER
PSL LIBRARY OBJECT SECTION
ERROR MESSAGES
COMPILER LISTING

G.4 COBOL Precompiler Input

G.4.1 General Information

This section describes the format of all inputs accepted by the precompiler. These inputs are in two separate data sets. One contains the option card and the second the structured ANS COBOL precompiler input. In the descriptions which follow, the word "condition" means any valid COBOL condition and "statement" means any valid COBOL statement. The basic formats of these input verbs are described in a meta-language which obeys the following notation:

a. In all formats, words in capital letters represent an actual occurrence of those words. If any such word is incorrectly spelled, it will not be recognized as a valid input and may cause an error in the program.

b. All underlined words are required unless the portion of the format containing them is itself optional. These are keywords. If any such word is missing or is incorrectly spelled, it is considered an error in the program.

c. Words that are printed in lower-case letters represent information to be supplied by the programmer. All such words are defined in the accompanying text.

d. Square brackets ([]) are used to indicate that the enclosed item may be used or omitted, depending on the requirements of the particular program.

e. The ellipsis (...) indicates that the immediately preceding lower-case word may occur once, or any number of times in succession.

f. Comments, restrictions, and clarifications on the use and meaning of every format are contained in the appropriate portions of the text.

In all the examples which follow, the code blocks are represented by the word "statement" in order to be consistent with the ANS COBOL standards manual. However, with the precompiler it is possible to place multiple sentences and even paragraphs between the structuring verbs if desired.

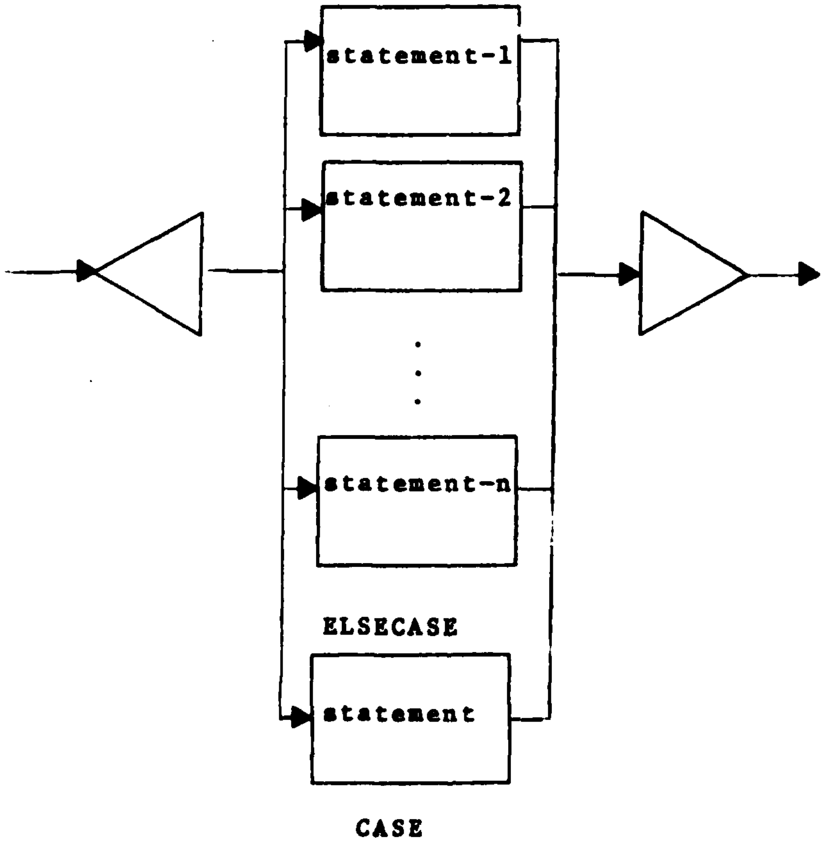
G.4.2 Precompiler Input Formats

Inputs discussed below are the control structure figures CASE, DOWHILE/DOUNTIL, IFTHENELSE, INCLUDE, and the unstructured code .ON/.OFF indicators.

G.4.2.1 CASE Figure

The CASE structure figure is used to pass control to one of a set of statements (or series of statements) depending on the value of an identifier.

The flowchart for the CASE control structure figure is:



and it is coded as:

Format

```
CASENTRY identifier

  [ CASE 1 [,numeric-literal-1]...  ]
  [                                 ]
  [   statement-1                   ]

  [ CASE 2 [,numeric-literal-2]...  ]
  [                                 ]
  [   statement-2                   ]

      .
      .
      .

  [ CASE n [,numeric-literal-n]...  ]
  [                                 ]
  [   statement-n                   ]

  [ ELSECASE     ]
  [              ]
  [   statement  ]

ENDCASE[.]
```

Control is transferred to one of a series of statements, depending on the value of identifier. When identifier has a value of 1 (or numeric-literal-1), control is passed to statement-1; a value of (or numeric-literal-2) causes control to be passed to statement-2; a value of n (or numeric-literal-n) causes control to be passed to statement-n. The identifier must represent an unsigned integer (i.e., 1,2,...,n). If the value of the identifier is within the range 1 through n but not equal to one of the CASE numbers, control is passed implicitly to the sentence following the ENDCASE. If the value of the identifier is outside of the range 1 through n, the ELSECASE statement, if one is present, is executed. Control is then passed implicitly to the sentence following the ENDCASE. CASE numbers need not be in numerical sequence. More than one CASE number may be associated with any CASE verb.

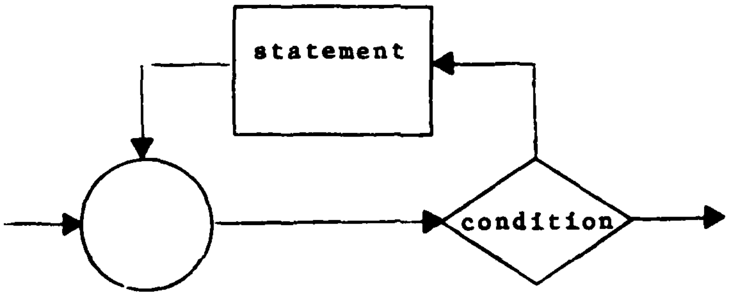
Statement/statement-n may consist of one or more statements and/or structured figures.

Identifier is the name of a numeric elementary item described as an integer. Its PICTURE must be of four digits or less. Its usage must be DISPLAY or COMPUTATIONAL.

G.4.2.2 DOWHILE/DOUNTIL Figures

The DO figures are used to depart from the normal sequence of procedures in order to execute a statement, or a series of statements, WHILE/UNTIL a predetermined condition is satisfied.

The flowchart for the DOWHILE control structure figure is:



DOWHILE

which is coded as:

Format

DO WHILE condition

    statement

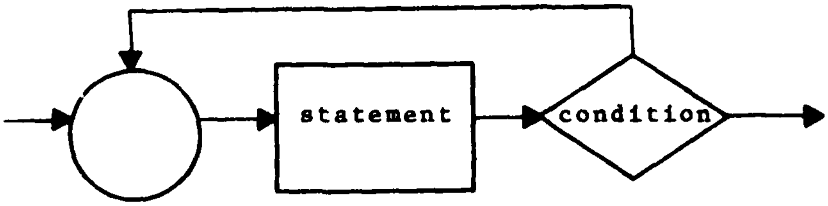
ENDDO[.]

When a DOWHILE is executed, the following action is taken:

a. As long as condition us tryem tge statement immediately following the condition (statement) is executed.

b. If condition is false, the next sentence which follows the ENDDO is executed.

The flowchart for the DOUNTIL control structure figure is:



DOUNTIL

and it is coded as:

Format

```
DO UNTIL condition

     statement

ENDDO[.]
```

When a DOUNTIL is executed, the following action is taken after the statement immediately following the condition (statement) is executed.

a. If condition is true, the next sentence which follows the ENDDO is executed.

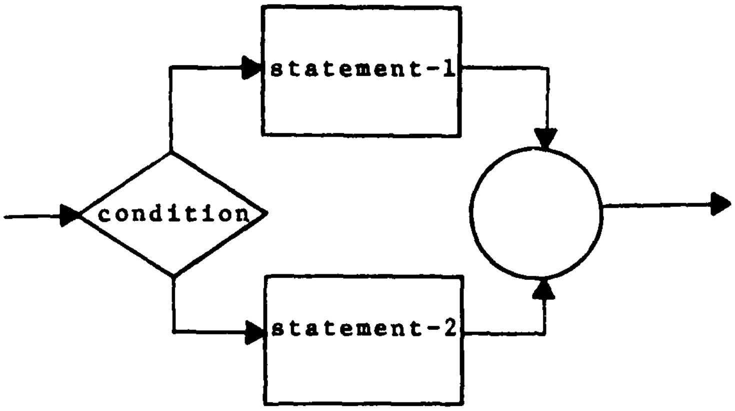
b. As long as condition remains false, the statement immediately following the condition (statement) is executed.

Statement in both DOWHILE and DOUNTIL control structure figures may consist of one or more statements and/or structure figures.

G.4.2.3 IFTHENELSE Figure

A conditional statement specifies that the truth value of a condition is to be determined and that the subsequent action of the object program is dependent on this truth value. The IFTHENELSE structure figure causes a condition to be evaluated with the subsequent action of the object program dependent upon whether the condition is true or false.

The flowchart for the IFTHENELSE control structure figure is:



IFTHENELSE

and the code structure to be used to represent the IFTHENELSE is:

Format

```
    IF condition

      statement-1

  [ ELSE                    ]
  [                         ]
  [   statement-2           ]

    ENDIF[.]
```

If an IFTHENELSE is executed, the following action is taken:

a. If condition is true, the statement immediately following the condition (statement-1) is executed. Control is then passed implicitly to the next sentence which follows the ENDIF.

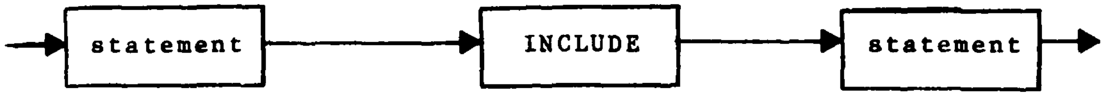
b. If condition is false, either the statement following ELSE (statement-2) is executed, or, if the ELSE option is omitted, the next sentence which follows the ENDIF is executed.

Statement-1 and statement-2 in the IFTHENELSE control structure figure map consist of one or more statements and/or structured figures. If a structured figure appears as statement-1 or statement-2, it is said to be nested. Nesting statements is much like specifying subordinate arithmetic expressions enclosed in parentheses for combination into larger arithmetic expressions. Indentation by PSL unit maintenance programs highlights both the structured figure and any nesting.

G.4.2.4 INCLUDE Figure

An INCLUDE statement is used to refer to a unit which will be retrieved and substituted in-line for the INCLUDE statement.

The flowchart for the INCLUDE figure is:



and the code structure to be used to represent the INCLUDE is:

Format

```
statement

numerics                INCLUDE            unit-name

statement
```

A more detailed description of INCLUDE statements is contained in Section 3.2.8.

G .4.2.5 Unstructured Code .ON/.OFF Indicators

It should be noted that none of the structuring verbs, except IF and ELSE, are contained as syntactical elements of ANS COBOL. Thus, there is no confusion when the precompiler processes these verbs for translation. However, this is not the case for the IF verb set. The precompiler IF requires a matching ENDIF whereas the COBOL IF does not. Thus, any IF statement which is processed by the precompiler is presumed to be a structuring verb and, if a corresponding ENDIF is not present, erroneous code may be generated. However, it is possible to pass unstructured code (particularly IF statements) through the precompiler by indicating the point at which structured processing is to be suspended and the point at which it is to be resumed. This is done by using the character strings .ON and .OFF as indicators to delimit the blocks which are not using the control structure figures.

The format of the precompiler structured/unstructured indicators is as follows:

Format
.ON

and

Format
.OFF

The .ON keyword indicates that all code which follows it, up to the next .OFF indicator, is unstructured and is not to be processed by the precompiler. The .OFF keyword indicates that the end of the unprocessed code has been reached and structured processing is to be reactivated. These keywords must appear on a single card (record) and must start in area A (columns 8 through 11).

G.5 Precompiler Output

G.5.1 General Information

When programming in a top-down manner using the INCLUDE capability, the statements in a small segment may be separated by hundreds of lines of code when the INCLUDEs are resolved. In order to facilitate the correspondence between the precompiler input and the translation to ANS COBOL for compiler input, each structured figure is assigned a unique number. This number is used in the generation of all paragraph-names associated with the complete verb set. This paragraph identification scheme is in addition to any optional mapping which might be requested.

Precompiler output consists of messages for detected errors and a translation of the COBOL input into a sequential ANS COBOL compiler compatible source program. This section is intended to describe the ANS COBOL generated by the precompiler when processing the COBOL precompiler input.

G.5.2 Precompiler Code Generation

G.5.2.1 CASE Figure

The CASE structure figure is used to pass control to one of a set of statements (or a series of statements) depending on the range of the identifier. If the value of the identifier is within the range $1 \leq$ identifier $\leq n$ (where n is the maximum CASE number) but not equal to one of the CASE numbers, control is passed to the ENDCASE collector. If the value of identifier is outside the range $1 \leq$ identifier $\leq n$, the ELSECASE statement is executed. The COBOL precompiler output for the CASE figure is constructed using a GO TO...DEPENDING ON and a single collector at the end of the figure. The code to represent the CASE figure is:

```
CASENTRY I
  CASE 4
    statement-1
  CASE 5,1
    statement-2
  ELSECASE
    statement
ENDCASE.
```

and is translated as:

```
1----CSENTRY.
    GO TO 1----CASETST.
1----CASE004.
        statement-1
    GO TO 1----ENDCASE.
1----CASE005.
1----CASE001.
        statement-2
    GO TO 1----ENDCASE.
1----CASETST.
    GO TO 1----CASE001 1----ENDCASE
            1----ENDCASE 1----CASE004
            1----CASE005
        DEPENDING ON I.
        statement
1----ENDCASE.
```

The generated paragraph-names for the CASE figure start in column 8 with a number (each use of any figure increments the number, which is limited to four digits, by 1) followed by dashes, and followed by the word CASENTRY, CASEnnn, CASETST, or ENDCASE, as appropriate, in order to form a 12-character paragraph-name.

G.5.2.2 DOWHILE/DOUNTIL Figures

The COBOL precompiler output for the DOWHILE figure is constructed using an IF statement and GO TO statements. The condition is tested prior to each execution of statement including the first. The code to represent the DOWHILE figure is:

```
           GO TO 1----DOTEST.
       1----DOWHILE.
             statement
       1-----DOTEST.
           IF (condition)
           GO TO 1----DOWHILE.
       1------ENDDO.
```

The COBOL precompiler output for the DOUNTIL figure is constructed using an IF statement and a GO TO statement. The condition is tested after each execution of statement, so that statement is always executed at least once. The test on the looping condition is negated by applying a NOT to the input condition. The code to represent the DOUNTIL figure is:

```
       1----DOUNTIL.
             statement
           IF NOT (condition)
           GO TO 1----DOUNTIL.
       1------ENDDO.
```

The generated paragraph-names for both the DOWHILE and DOUNTIL figures start in column 8 with a number (each use of any figure increments the number, which is limited to four digits, by 1), followed by dashes, and followed by the word DOUNTIL, DOWHILE, DOTEST, or ENDDO, as appropriate, in order to form a 12-character paragraph-name.

G.5.2.3 IFTHENELSE Figure

The COBOL precompiler output for the IFTHENELSE figure is so constructed that the "true" conditional code, statement-1, immediately follows the IF sentence. In order to implement this in COBOL, it is necessary to test for the negative of the condition by using a NOT (a positive test would necessitate the use of an additional GO TO). The code to represent the IFTHENELSE figure is:

```
1---------IF.
    IF NOT (condition)
    GO TO 1-------ELSE.
      statement-1
    GO TO 1------ENDIF.
1-------ELSE.
      statement-2
1------ENDIF.
```

The generated paragraph-names start in column 8 with a number (each use of any figure increments the number, which is limited to four digits, by 1), followed by dashes, and followed by the word ELSE or ENDIF, as appropriate, in order to form a 12-character paragraph-name.

The ELSE in the IFTHENELSE figure is optional and if it is not used, the code generated is:

```
1---------IF.
    IF NOT (condition)
    GO TO 1-------ELSE.
      statement-1
1-------ELSE.
1------ENDIF.
```

The GO TO, which is generated when the IF verb is detected, is always to the ELSE paragraph-name in anticipation of the presence of one. Therefore, the ELSE paragraph-name must be generated even when ELSE code is not present.

G.6 COBOL Compiler Restrictions

G.6.1 Reserved Words

The following additional words are added to the ANS COBOL reserved word list when using the precompiler:

```
CASE
CASENTRY
DO
ELSECASE
ENDCASE
ENDDO
INCLUDE
ENDIF
WHILE
UNTIL
```

G.6.2 Paragraph-Names

In order to insure uniqueness of paragraph-names, the precompiler user may not code any such names in the format generated by the precompiler. A description of the format follows. All paragraph-names are 12-characters long. They start with a one to four digit number and terminate with one of the following character strings:

```
CASEnnn (nnn is a number padded to 3 digits
         with leading zeroes if required)
CASETST
CSENTRY
DOTEST
DOUNTIL
DOWHILE
ELSE
ENDCASE
ENDDO
ENDIF
IF
```

Separating the leading digit and the terminating character string are as many dashes as required to pad the paragraph-name to a length of 12 characters.

G.6.3 Structuring Verb Sets

As mentioned in Section G.2, the precompiler supports three verb sets, namely, IF, DO, and CASE. These sets have starting verbs (IF, DO, CASENTRY) and ending verbs (ENDIF, ENDDO, ENDCASE) and are designed to be nested so that between any two members of a set, another complete set may be inserted, thus increasing the nesting depth by one. A complete set is defined as one having both a starting and ending verb.

It is not permissible to intersperse a verb from one set in the middle of another set. Thus, the sequence of verbs:

```
IF
 DO
 ENDDO
ELSE
ENDIF
```

is a valid nesting, but:

```
IF
 DO
 ELSE
 ENDDO
ENDIF
```

is not. At any nest level, as many sets as required may be used, provided one set is completed before the next one is started.

G.6.4 Other Restrictions

The following additional rules must be followed by precompiler users:

a. All PERFORMs must be written as PERFORM paragraph-name-1 THRU paragraph-name-2.

b. Every COBOL verb (as well as control structure verbs) must appear at the start of a new line.

c. All control structure figures and the INCLUDE statement as defined by the structure verbs are to be considered conditional statements. They may not be used where ANS COBOL syntax calls for an imperative statement such as in an arithmetic statement following ON SIZE ERROR, in READ/WRITE statements after INVALID KEY, following the AT END or WHEN in a SEARCH statement, and other such occurrences.

d. Whenever a structuring verb is detected by the precompiler, a period is added to the statement preceding it if one is not already present. Thus, if the previous statement was a COBOL conditional statement, the presence of any structuring verb terminates it.

e. Any code inserted in-line by a COPY statement will not be scanned by the precompiler.

G.7 Error Messages

The precompiler error messages are divided into two categories --messages which result from an error condition under which the precompiler continues processing, and messages which result from conditions under which processing is terminated. The two categories are identified by separate paragraphs.

G.7.1 Errors Which Do Not Terminate Processing

When errors which result in the generation of any messages in this subsection are detected, the precompiler will attempt to take corrective action where possible. However, in some cases such action may be wrong and result in improper execution. Therefore, the programmer should always correct his code to eliminate these messages.

MSG 111 UNIT unit-name LINE line-nbr
CASE ENTRY STATEMENT MISSING NUMBER IDENTIFIER.
ERRONEOUS CODE MAY RESULT.

The identifier on the CASENTRY verb could not be found. It will therefore be missing when the GO TO...DEPENDING ON is generated.

MSG 112 UNIT unit-name LINE line-nbr
DO STATEMENT MISSING WHILE OR UNTIL.
WHILE ASSUMED.

A WHILE or UNTIL was not detected after a DO.

MSG 113 UNIT unit-name LINE line-nbr
CASE FIGURE HAS NO CASE.

The precompiler did not find any CASE verbs. The GO TO...DEPENDING ON is not generated in this instance. If an ELSECASE is present, control falls into this code.

MSG 114 UNIT unit-name LINE line-nbr
EXTRA ELSECASE FOUND FOR CASE FIGURE DISCARDED.

The precompiler detected more than one ELSECASE as part of a CASE statement. In this instance control falls into the extra ELSECASE code.

MSG 115 UNIT unit-name LINE line-nbr
CASE STATEMENT FOUND AFTER ELSECASE WILL NOT BE EXECUTABLE.

The code comprising the misplaced CASE is unreachable.

MSG 116 UNIT unit-name LINE line-nbr
EXTRA ELSE STATEMENT FOR IF FIGURE DISCARDED.

The precompiler has detected more than one ELSE for the same IF. Control falls into the extra ELSE code.

MSG 117 UNIT unit-name LINE line-nbr
new-figure ENCOUNTERED BEFORE IF FIGURE TERMINATED. ENDIF ASSUMED PRECEDING new-figure.

This error message is generated whenever an intermediate or terminating structure verb, which is not part of the lowest level verb set that is currently open, has been detected. Before the message is selected for output, it is first determined that a starting verb does exist at some higher nesting level. An end of file condition (EOF) causes a similar action. The lines at the beginning and end of each message will contain the verb or (EOF) that does not match the lowest level open verb set. The precompiler assumes that the proper terminator for that nesting level was omitted. It therefore terminates the figure at that nest level and then examines the next figure within which the terminated figure was nested. This termination procedure continues until the nesting level is reached with the proper starting verb. That figure is then closed and normal processing is resumed.

MSG 118 UNIT unit-name LINE line-nbr
new-figure ENCOUNTERED BEFORE DO FIGURE TERMINATED. ENDDO ASSUMED PRECEDING new-figure.

This error message is generated whenever an intermediate or terminating structure verb, which is not part of the lowest level verb set that is currently open, has been detected. Before the message is selected for output, it is first determined that a starting verb does exist at some higher nesting level. An end of file condition (EOF) causes a similar action. The lines at the beginning and end of each message will contain the verb or (EOF) that does not match the lowest level open verb set. The precompiler assumes that the proper terminator for that nesting level was omitted. It therefore terminates the figure at that nest level and then examines the next figure within which the terminated figure was nested. This termination procedure continues until the nesting level is reached with the proper starting verb. That figure is then closed and normal processing is resumed.

MSG 119 UNIT unit-name LINE line-nbr
new-figure ENCOUNTERED BEFORE CASE FIGURE
TERMINATED. ENDCASE ASSUMED PRECEDING new-figure.

This error message is generated whenever an intermediate or terminating structure verb, which is not part of the lowest level verb set that is currently open, has been detected. Before the message is selected for output, it is first determined that a starting verb does exist at some higher nesting level. An end of file condition (EOF) causes a similar action. The lines at the beginning and end of each message will contain the verb or (EOF) that does not match the lowest level open verb set. The precompiler assumes that the proper terminator for that nesting level was omitted. It therefore terminates the figure at that nest level and then examines the next figure within which the terminated figure was nested. This termination procedure continues until the nesting level is reached with the proper starting verb. That figure is then closed and normal processing is resumed.

MSG 120 UNIT unit-name LINE line-nbr
UNMATCHED new-figure DELETED.

An intermediate or terminating structuring verb cannot be matched with its starting verb. The intermediate or terminating verb is ignored.

MSG 121 UNIT unit-name LINE line-nbr
EXPECTED CASE NUMBER. FOUND WORD user-word.
WORD DISCARDED.

A non-numeric value for a CASE verb was detected. The word that was found, inserted where the blank line is, is ignored by the program.

MSG 122 UNIT unit-name LINE line-nbr
OPTION CARD input-card-image IN ERROR
DEFAULT OPTIONS - NOSOURCE,MAP - ASSUMED.

The entire 80 column input card is printed where input-card-image is indicated.

G.7.2 Errors Which Terminate Processing

The error messages which follow are ones which suspend precompiler processing. Six of them are caused by exceeding the size of the various push down stacks used by the precompiler to store information. In some of the cases, if the program cannot be decreased in size, it may be necessary to increase the size of the stack which overflowed and recompile the precompiler.

MSG 123 UNIT unit-name LINE line-nbr
NEST-STACK OVERFLOWED. MAXIMUM NUMBER OF
NESTED STRUCTURE FIGURES 50 EXCEEDED. FATAL
ERROR. EXECUTION TERMINATED.

This stack contains the starting verb of each verb set at each nested level. At any one time in the program, the maximum depth to which figures may be nested is currently set at 50.

MSG 124 UNIT unit-name LINE line-nbr
CONDITION-STACK OVERFLOWED. NESTED DO
CONDITIONS OCCUPY MORE THAN MAXIMUM 50 LINES.
FATAL ERROR. EXECUTION TERMINATED.

The conditionals on DO statements are saved in a stack and removed when the corresponding ENDDO is detected. The stack is set to hold 50 cards maximum.

MSG 125 UNIT unit-name LINE line-nbr
CASE-STACK OVERFLOWED. MAXIMUM NUMBER OF NESTED CASES 10 EXCEEDED. FATAL ERROR. EXECUTION TERMINATED.

This stack controls the depth to which one CASE figure may be nested within other CASE figures. The current maximum is set at 10.

MSG 126 UNIT unit-name LINE line-nbr
CASE-LABEL-STACK OVERFLOWED. MAXIMUM NUMBER OF CASE NUMBERS 200 EXCEEDED. FATAL ERROR. EXECUTION TERMINATED.

This stack is currently set to hold a maximum of 200 CASE numbers. This is not meant to imply that only 200 CASE numbers may be present in a given program since once a CASE statement is terminated with an ENDCASE, the CASE numbers associated with it are removed from the stack.

MSG 127 UNIT unit-name LINE line-nbr
LABEL-STACK OVERFLOWED. MAXIMUM LABELS 125 EXCEEDED. CAUSED BY TOO MANY NESTED STRUCTURE PICTURES. FATAL ERROR. EXECUTION TERMINATED.

The LABEL-STACK holds the paragraph-names which are generated by the precompiler. The number of such names in the stack at any one time is a function of the depth of nesting at that point in the program. The current maximum is set to 125.

MSG 128 UNIT unit-name LINE line-nbr
OUTPUT-STACK OVERFLOWED. PROBABLY CAUSED BY TOO MANY BLANK OR CONTINUATION LINES FOR ONE STATEMENT. FATAL ERROR. EXECUTION TERMINATED.

OUTPUT-STACK is the area within which all processing is carried on. Input records are saved in this stack while they are analyzed. Generated paragraph-names and COBOL statements are also placed in this stack prior to being directed t the output data set. This stack is normally unloaded prior to the analysis of the next COBOL input record. The current maximum is set to 50.

MSG 129 UNIT unit-name LINE line-nbr
PROCEDURE DIVISION NOT FOUND. FATAL ERROR.
EXECUTION TERMINATED.

The precompiler was unable to locate the PROCEDURE DIVISION in order to start its processing.

MSG 130 UNIT unit-name LINE line-nbr
MAXIMUM NUMBER OF STRUCTURE FIGURES 9999
EXCEEDED. FATAL ERROR. EXECUTION TERMINATED.

The digital value placed at the front of all precompiler generated paragraph-names has been exceeded.

APPENDIX H. USERS MANUAL FOR PSL STRUCTURED JOVIAL PRECOMPILER

The following pages of this appendix provide the relevent information on the JOCIT/JOVIAL Precompiler* of the PSL. This precompiler was designed for the JOVIAL language augmented with structured figures. The JOVIAL language is JOVIAL-J-3 as defined in the Air Force's document AFM 100-24 with modifications, as indicated in the JOCIT Compiler's User's Manual. The JOCIT/JOVIAL Precompiler was developed to process structured programming figures in accordance with the standards in Programming Language Standards, Volume I of the Structured Programming Series, RADC-74-200.

H.1 Precompiler Objectives

The objective of the JOVIAL precompiler is to assist the user in writing this language in a structured format. This is achieved by defining a precompiler syntax which introduces into the JOVIAL language the statements for writing structured CASE, DOWHILE/DOUNTIL and IFTHENELSE figures. The precompiler translates these statements into equivalent expressions in standard JOVIAL for compilation by a JOCIT/JOVIAL compiler. An INCLUDE statement is also introduced into the precompiler syntax, which is used to refer to a unit which will be retrieved and substituted in-line for the INCLUDE statement.

H.2 Precompiler Inputs

The JOVIAL precompiler accepts as input a JOVIAL program written in precompiler syntax as defined in Section H.4. The structuring verbs described in that section are processed by algorithms to produce the output indicated in Section H.5. These structured figures should be viewed as complete sets which are designed to be processed as units. Four such sets are processed with this precompiler. They are:

a. The IF set

- IF
- ELSE (optional)
- ENDIF

*For additional information regarding this capability as it was developed, see the Structured Programming JOCIT/JOVIAL Precompiler produced under RADC contract F30602-76-C-0166 by IBM.

b. The DO sets

- DO (WHILE or UNTIL)
- ENDDO

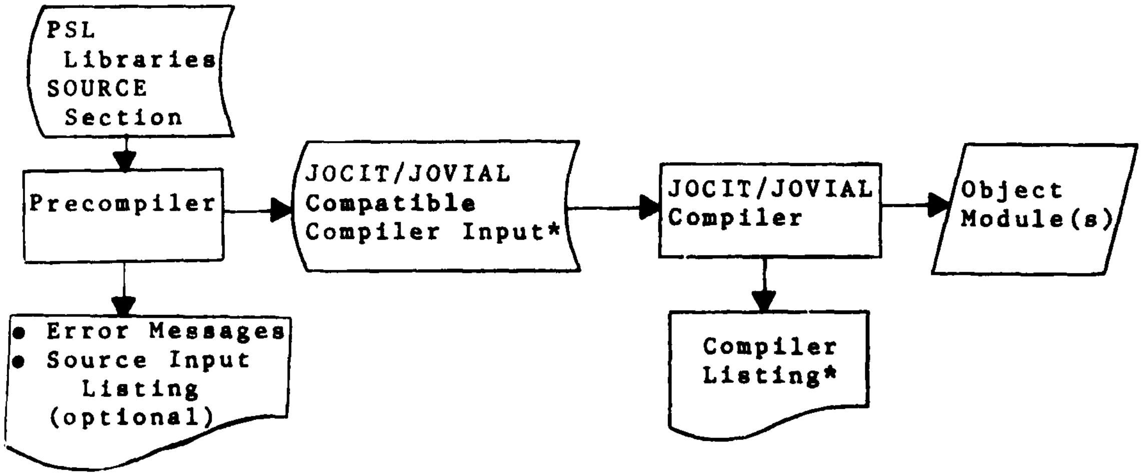
c. The CASE set

- CASENTRY
- CASE (at least one must be present)
- ELSECASE (optional)
- ENDCASE

H.3 Precompiler Output

The primary output of the precompiler is JOCIT/JOVIAL compatible compiler input. This data set, input to the JOCIT/JOVIAL compiler, is different from the compiler source code listing because of the intermediate translation. The precompiler can produce as an optional output, a listing of the JOVIAL input with assigned sequence numbers associated with each statement on the source listing. By exercising a second option, the user may request that the sequence numbers on this source listing be inserted in columns 73 through 80 of the precompiler output (compiler input) and subsequently the compiler output listings. Thus, the programmer can relate a statement on the compiler output listing to the corresponding precompiler input. Finally, a third precompiler output may consist of error messages (Section H.7), should any errors be detected.

The relationship between the precompiler and its inputs and outputs is graphically displayed by the following:


PSL Libraries SOURCE Section
Precompiler
JOCIT/JOVIAL Compatible Compiler Input*
JOCIT/JOVIAL Compiler
Object Module(s)
● Error Messages
● Source Input Listing (optional)
Compiler Listing*

*Optional source input listing sequence numbers in columns 73-80.

H.4 JOVIAL Precompiler Input

H.4.1 General Introduction

This section defines the formats for all the precompiler inputs. These inputs are in two separate data sets. One contains the OPTION CARD and the second, the structured JOVIAL precompiler input. The basic formats of the input verbs for the JOCIT/JOVIAL precompiler are described in a meta-language defined in the JOVIAL J3 program language manual. The applicable subset of this meta-language which is used in this JOVIAL subsection follows:

a. In all formats, words in capital letters represent occurrence of those words. If any such word is incorrectly spelled, it will not be recognized as a valid input and may cause an error in the program.

b. Words that are printed in lower-case letters represent information to be supplied by the programmer. All such words are JOVIAL defined words.

c. In the descriptive meta-language for JOVIAL, the colon is used to connect several names into a single composite name. Such names -- simple and composite -- are meta-linguistic elements. "Boolean:formula" is an example of this notation.

d. Brackets ([ ]) are used to indicate that the enclosed item may be used or omitted.

e. The vertical elipsis (⋮) is used to in the CASE structure to indicate that the CASE verb and associated statement may occur once, or any number of times in succession.

f. The special symbol θ is used to indicate that one or more spaces are permitted.

g. Comments, restrictions, and clarifications on the use and meaning of every format are contained in the appropriate portions of the text.

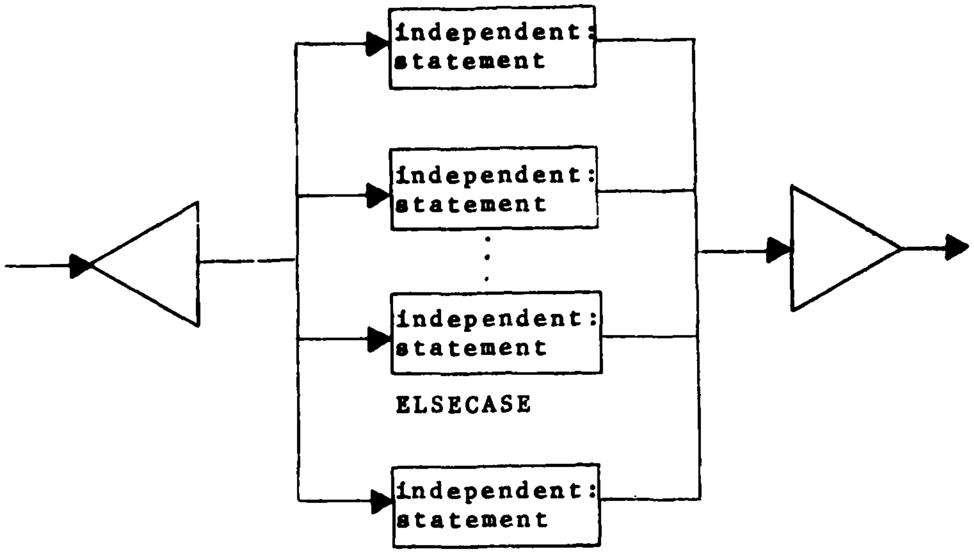
H.4.2 Precompiler Input Formats

Inputs discussed below are the control structure figures CASE, DOWHILE/DOUNTIL, IFTHENELSE, INCLUDE, and the OPTION-CARD.

H.4.2.1 CASE Figure

The CASE:figure is used to pass control to one of a set of statements depending on the value of a numeric:formula.

The flowchart for the CASE:figure is:



CASE

and it is coded as:

```
    CASENTRY θ numeric:formula θ $

      CASE θ numeric:constant:list $

          independent:statement

    ┌ CASE θ numeric:constant:list $┐
    └     independent:statement     ┘

      .
      .
      .
    ┌ CASE θ numeric:constant:list $┐
    └     independent:statement     ┘

    ┌ ELSECASE                ┐
    └     independent:statement┘

    ENDCASE
```

The numeric:constant:list consists of one or more numeric :constants separated by commas.

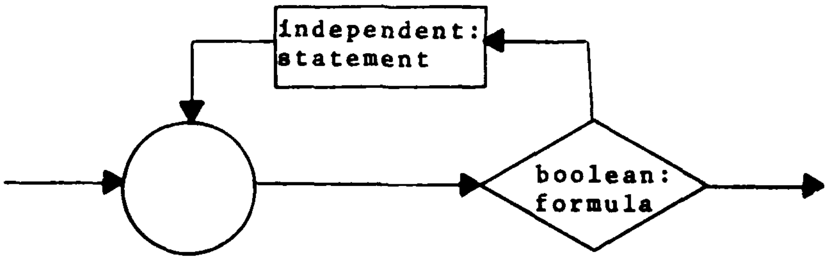
In executing the CASE:figure, the numeric:formula following the CASENTRY is evaluated as an integer. The independent :statement associated with the CASE number corresponding to the value of the numeric:formula is executed, and control is then passed to the statement following the ENDCASE. If the numeric:formula yields a positive value that does not correspond to a CASE numeric:constant in the list but with a value less than the maximum numeric:constant, the execution sequence continues with the next statement following the ENDCASE. If the numeric:formula yields a negative value or zero, or a positive value greater than the maximum numeric :constant in the CASE:figure, the independent:statement associated with the ELSECASE is executed.

After execution of any statement in list, the execution sequence continues with the next statement following the ENDCASE.

BEGIN and END brackets within a CASE are not required when its independent:statement is a compound:statement.

H.4.2.2 DOWHILE/DOUNTIL Figures

The DO figures allow iterative execution of an independent :statement based on the value of a Boolean:formula. The value of the Boolean:formula is tested prior to the execution of the independent:statement in a DOWHILE:figure and after the execution of the independent:statement in a DOUNTIL:figure. The flowchart for the DOWHILE:figure is:



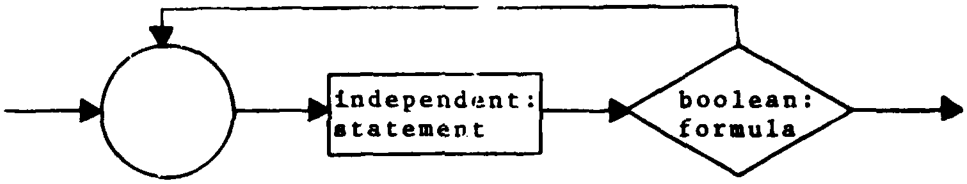
DOWHILE

which is coded as:

```
DO WHILE θ Boolean:formula θ $

  independent:statement

ENDDO
```

The DOWHILE:figure executes the independent:statement that physically follows the DOWHILE:statement as long as the value of the Boolean:formula is true. The Boolean:formula is tested prior to the execution of any statements within the range of the DOWHILE:figure (i.e., the statements that physically follow the DOWHILE, up to and including the statement preceding the ENDDO). BEGIN and END brackets are not required when the independent:statement is a compound:statement.

The flowchart for the DOUNTIL:figure is:



DOUNTIL

and it is coded as:

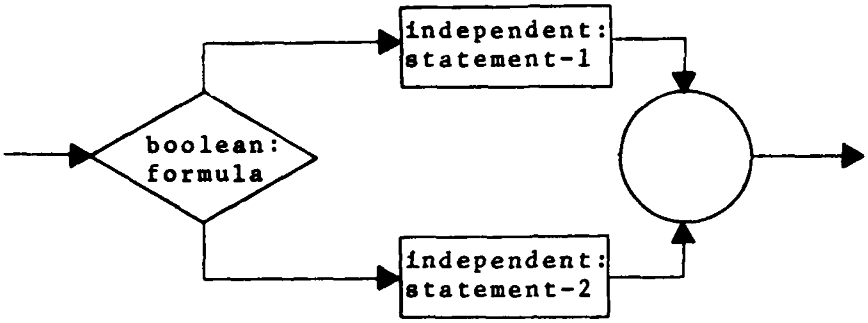
```
DO UNTIL θ Boolean:formula θ $

  independent:statement

ENDDO
```

The DOUNTIL:figure executes the independent:statement that physically follows the DOUNTIL:statement as long as the value of the Boolean:formula is false. The Boolean:formula is tested after the execution of any statements within the range of the DOUNTIL:figure (i.e., the statements that physically follow the DOUNTIL, up to and including the statement preceding the ENDDO). BEGIN and END brackets are not required when the independent:statement is a compound:statement.

H.4.2.3 IFTHENELSE Figure

The IFTHENELSE:figure is a structured conditional statement providing for the conditional execution of one of two independent:statements based upon the value of a Boolean:formula.

The flowchart for the IFTHENELSE:figure is:



IFTHENELSE

and the code structure to be used to represent the IFTHENELSE :figure is:

```
   IF θ Boolean:formula θ $

     independent:statement-1

  [ELSE                           ]
  [  independent:statement-2      ]

   ENDIF
```

In either position in the definition of the IFTHENELSE:figure, independent:statement is a simple:statement, a compound :statement, or a structured:figure. A structured:figure is an IFTHENELSE:figure, a DOWHILE:figure, a DOUNTIL:figure, or a CASE:figure.

The effect of an IFTHENELSE:figure is that if the value of the Boolean:formula is true, independent:statement-1 which follows it is executed; otherwise, independent:statement-2 following the ELSE is executed. BEGIN and END brackets are not required when the independent:statement is a compound:statement.

H.4.2.4 INCLUDE Figure

An INCLUDE statement is used to refer to a unit which will be retrieved and substituted in-line for the INCLUDE statement.

The flowchart for the INCLUDE figure is:

statement
INCLUDE
statement

and the code structure to be used to represent the INCLUDE is:

```
                                FORMAT
    statement

    INCLUDE              unit-name

    statement
```

A more detailed description of INCLUDE statements is contained in Section 3.2.8.

H.4.2.5 Option Card

The option card is a single 80-character input record which contains keywords to indicate whether a listing of the precompiler JOVIAL input is to be generated. This listing is intended to assist the programmer in making a correlation between the compiler source code listing and the precompiler JOVIAL input. The selection of options is contingent upon the environment in which the program is being developed.

The card is optional. Its format is:

```
                                Format

   [OPTION     [,]]      [SOURCE   ] [[,]]      [MAP  ]
   [               ]      [NOSOURCE ]            [NOMAP]
```

While the format description indicates SOURCE/NOSOURCE keywords precede MAP/NOMAP keywords, the mutually exclusive keywords SOURCE/NOSOURCE and MAP/NOMAP may precede or follow each other. Their appearance is restricted to columns 1 through 72 of the option card and they may be separated from each other by a comma or blank. The SOURCE keyword controls the option as to whether a sequenced source listing of the precompiler input is to be produced. The MAP keyword indicates that the sequence numbers on the optional precompiler source listing are to be placed in columns 73 through 80 of the precompiler output. Default options will cause a precompiler source listing and will sequence the precompiler output data set.

H.5 Precompiler Output

H.5.1 General Information

When programming in a top down manner using the INCLUDE capability, the statements in a small segment may be separated by hundreds of lines of code when the INCLUDEs are resolved. In order to facilitate the correspondence between the precompiler input and the translation to standard JOVIAL for compiler input, each structured figure is assigned a unique number. This identification scheme is in addition to any optional mapping which might be requested.

In general, precompiler output consists of messages for detected errors, a translation of the source input into a compiler compatible source program, and an optional sequenced listing of the precompiler input.

H.5.2 Precompiler Code Generation

H.5.2.1 CASE Figure

The CASE:figure is used to pass control to one of a set of statements depending on the value of a numeric:formula. If the value of the numeric:formula is within the range 1   numeric:formula, n (where n is the maximum CASE number) but not equal to one of the CASE number, control is passed to the ENDCASE collector. If the value of numeric:formula is outside the range 1   numeric:formula   n, the ELSECASE independent:statement, is executed; otherwise, control is passed to the ENDCASE collector. The JOCIT/JOVIAL precompiler output for the CASE:figure should be constructed around a switch:declaration. The code generated by the JOVIAL precompiler to represent this CASE:figure:

```
CASENTRY NUMB $
  CASE 5,1 $
    ALPHA = 0.5 $
    BETA = ALPHA**2 $
  CASE 4 $
    ALPHA = 0.0 $
  ELSECASE
    ALPHA = 1.0 $
    BETA = 1.0 $
ENDCASE
```

should be:

```
CASENTRY'1 = NUMB $
  GOTO SWITCH'1 $
  CASE'1'5.
  CASE'1'1.
    ALPHA = 0.5 $
    BETA = ALPHA**2 $
    GOTO ENDCASE'1 $
  CASE'1'4.
    ALPHA = 0.0 $
    GOTO ENDCASE'1 $
  ELSECASE'1.
    ALPHA = 1.0 $
    BETA = 1.0 $
    GOTO ENDCASE'1 $
  SWITCH'1.
    IF CASENTRY'1 LS 1 OR
       CASENTRY'1 GR 5 $
       GOTO ELSECASE'1 $
    GOTO SW'1 $ SW'1.
    SWITCH CASENTRY'1 = (,CASE'1'1,,,
      CASE'1'4,CASE'1'5) $
ENDCASE'1.
```

In the example above, a value of 1 or 5 for NUMB causes the code with the statement:names CASE'1'5 and CASE'1'1 to be executed with ALPHA set equal to 0.5 and BETA set equal to the square of ALPHA. A value of 4 for NUMB causes CASE'1'4 to be executed with ALPHA set equal to zero. A value of 2 or 3 for NUMB causes control to be passed directly to the statement following the ENDCASE'1 statement:name. If NUMB is outside the range $1 \leq$ NUMB ≤ 5, the ELSECASE'1 code is executed with both ALPHA and BETA set equal to 1.0.

If the EL[illegible]ASE had been omitted on input, the generated code would be:

```
CASENTRY'1 = NUMB $
  GOTO SWITCH'1 $
  CASE'1'5.
  CASE'1'1.
    ALPHA = 0.5 $
    BETA = ALPHA**2 $
    GOTO ENDCASE'1 $
  CASE'1'4.
    ALPHA = 0.0 $
    GOTO ENDCASE'1 $
  SWITCH'1.
    SWITCH CASENTRY'1 = (,CASE'1'1,,,
      CASE'1'4,CASE'1'5) $
ENDCASE'1.
```

The generated statement:names SWITCH'n, SW'n, and ENDCASE'n, should start with SWITCH, SW, and ENDCASE, respectively, be followed by a prime ('), and an unsigned integer:constant without a scale. The integer:constant should begin at 1 and be incremented by 1 for each occurrence of a structured:figure. The switch:name should start with CASENTRY, be followed by a prime ('), and also be followed by the CASE:figure's unique integer:constant. The CASE statement:names should start with CASE, be followed by a prime ('), the CASE:figure's unique integer:constant, another prime ('), and finally, by the appropriate integer:constant CASE number (unique within a CASE:figure). Indentation of generated code should be as shown.

Since the index value of the index:switch:declaration points out the required sequence:designator according to its position in the list, starting with zero and not one, the index:switch: list will always begin with a comma, since a zero CASE is not permitted. Commas without corresponding sequence:designators indicate case values which were not defined in the input CASE:figure.

H.5.2.2 DOWHILE/DOUNTIL Figure

The JOCIT/JOVIAL precompiler output for the DOWHILE:figure should be constructed using an if:clause and go:to:statements. The Boolean:formula is tested prior to each execution of independent:statement including the first. The code to represent the DOWHILE:figure is:

```
GOTO ENDDO'1 $
DOWHILE'1.
  independent:statement
ENDDO'1. IF Boolean:formula $
         GOTO DOWHILE'1 $
```

The JOCIT/JOVIAL precompiler output for the DOUNTIL:figure should be constructed using an if:clause and a go:to:statement. The Boolean:formula is tested after each execution of independent:statement, so that independent:statement is always executed at least once. The test on the looping Boolean:formula should be negated by applying a NOT to the input Boolean:formula. The code generated by the JOVIAL precompiler to represent the DOUNTIL:figure is:

```
DOUNTIL'1.
  independent:statement
  IF NOT (Boolean:formula) $
  GOTO DOUNTIL'1 $
ENDDO'1.
```

The generated statement:names for both the DOWHILE:figure and DOUNTIL:figure should start with DOWHILE, DOUNTIL, and ENDDO, respectively, be followed by a prime (') and an unsigned integer:constant without a scale. The integer :constant for each shoula begin at 1 and be incremented by 1 for each appearance of a structured:figure to create unique statement:names.

The DOWHILE and ENDO statement:names should be aligned as well as the DOUNTIL and ENDDO statement:names. Generated statements within the figures should be aligned as indicated in the above examples.

H.5.2.3 IFTHENELSE Figure

The IFTHENELSE:figure provides for the conditional execution of one of two independent:statements based upon the value of a Boolean:formula. In order to avoid the necessity of using BEGIN and END brackets for the independent:statements and to avoid the use of a null ORIF when the optional ELSE clause is not used, the code generated by the JOVIAL precompiler to represent the IFTHENELSE:figure is:

```
IF NOT (Boolean:formula) $
  GOTO ELSE'1 $
  independent:statement-1
  GOTO ENDIF'1 $
ELSE'1.
  independent:statement-2
ENDIF'1.
```

The generated IF, ELSE'n, and ENDIF'n should be aligned as were the original (input) IF, ELSE, and ENDIF.

The ELSE in the IFTHENELSE:figure is optional and if it is not used, the code to represent this is:

```
IF NOT (Boolean:formula) $
  GOTO ELSE'1 $
  independent:statement-1
ELSE'1.
ENDIF'1.
```

The generated statements should be aligned as indicated above.

The generated statement:names in both of the above examples should start with ELSE and ENDIF, respectively, be followed by a prime (') and an unsigned integer:constant without a scale. The integer:constant should begin at 1 and be incremented by 1 for each appearance of a structured programming figure to create unique statement:names.

H.6 JOVIAL Precompiler Restrictions

H.6.1 Primitives

The following additional words are added to the JOCIT/JOVIAL primitive list when using the precompiler:

CASE
CASENTRY
DO
ELSE
ELSECASE
ENDCASE
ENDDO
ENDIF
UNTIL
WHILE.

H.6.2 Statement Names and Variable Names

In order to insure uniqueness of statement names and variable names the precompiler user must avoid coding names in the format generated by the precompiler. A list of the formats follows:

STATEMENT NAMES

```
     CASE'NNN'MMMMMM      (NNN and MMMMMM are respectively,
                          3 and 6 digit numbers with loading
                          zeroes intact)
     DOUNTIL'MMMMMM
     DOWHILE'MMMMMM
     ELSE'MMMMMM
     ELSECASE'MMMMMM
     ENDCASE'MMMMMM
     ENDDO'MMMMMM
     ENDIF'MMMMMM
     SW'MMMMMM
     SWITCH'MMMMMM
```

VARIABLE NAMES

```
     CASENTRY'MMMMMM.
```

H.6.3 Structured Figures

As mentioned in Section 2, the precompiler supports four structured figures, namely, IFTHENELSE, DOUNTIL, DOWHILE, and CASE. These figures have starting names (IF, DO UNTIL, DO WHILE, CASENTRY) with corresponding ending names (ENDIF, ENDDO, ENDDO, ENDCASE). These figures may be nested. That is, the independent statements within any structured:figure may also contain completed structured figures. Any complete structured figure contained within another is said to be at a nesting depth which is 1 greater than the containing figure.

It is not permissible to overlap figures -- they must be nested. Thus, the sequence:

```
IF
  DO
  ENDDO
ELSE
ENDIF
```

is a valid nesting, but:

```
IF
  DO
  ELSE
  ENDDO
ENDIF
```

is not.

H.6.4 Other Restrictions

The following additional rules must be followed by precompiler users:

a. Every structure:word (IF, ELSE, ENDIF, CASENTRY, CASE, ELSECASE, DO, ENDDO) must appear as the first symbol on the input record.

b. Symbols INCLUDE, DIRECT and JOVIAL, when used, must also appear as the first symbol on the input record.

c. The IF statement as defined in standard JOVIAL must not be used. Instead, use the structure:figure IF.

d. GOTO statements should not be used except when used as GOTO CLOSE:name.

H.7 Error Messages

H.7.1 General Information

The precompiler error messages are divided into two categories. Those messages whose numbers are less than 100 are errors which are detected but for which the precompiler continued processing. Those with numbers in the one hundred range are errors for which the precompiler processing is terminated at the point the error was detected. In this case the compilation should also be suspended. However, the method for suspending compilations is a system dependent function and not addressed by the precompiler.

H.7.2 Messages

H.7.2.1 Errors Which Do Not Terminate Processing

When errors which result in the generation of any messages in this subsection are detected, the precompiler will attempt to take corrective action where possible. However, in many cases such action may be wrong and result in improper program execution. Therefore, the programmer should always correct his code to eliminate these messages.

MSG 002 DO MISSING WHILE OR UNTIL. WHILE ASSUMED.

A WHILE or UNTIL was not detected after a DO.

MSG 003 CASE FIGURE HAS NO CASE.

The precompiler did not find any CASE verbs. If an ELSECASE is present, control falls into this code.

MSG 004 EXTRA ELSECASE FOUND FOR CASE FIGURE DISCARDED.

The precompiler detected more than one ELSECASE as part of a case statement. In this instance control falls into the extra ELSECASE code.

MSG 005 CASE STATEMENT FOUND AFTER ELSECASE DISCARDED.

The code comprising the misplaced CASE is unreachable.

MSG 006 EXTRA ELSE STATEMENT FOR IF FIGURE DISCARDED.

The precompiler has detected more than one ELSE for the same IF. Control falls into the extra ELSE code.

MSG 007 ________________ENCOUNTERED BEFORE IF FIGURE TERMINATED. ENDIF ASSUMED PRECEDING ___________.

This error message is generated whenever an intermediate or terminating structure word that is not part of the lowest level structure that is currently open has been detected. Before the message is selected for output it is first determined that a starting word does exist at some higher nesting level. An end of file condition (EOF) causes a similar action. The lines at the beginning and end of each message will contain the word or (EOF) that does not match the lowest level open structure. The precompiler assumes that the proper terminator for that nesting level was omitted. It therefore terminates the figure at that nest level and then examines the next figure within which the terminated figure was nested. This termination procedure continues until the nesting lėvel is reached with the proper starting word. That figure is then closed and normal processing is resumed.

MSG 008 ________________ENCOUNTERED BEFORE DO FIGURE TERMINATED. ENDDO ASSUMED PRECEDING __________.

This error message is generated whenever an intermediate or terminating structure word that is not part of the lowest level structure that is currently open has been detected. Before the message is selected for output it is first determined that a starting word does exist at some higher nesting level. An end of file condition (EOF) causes a similar action. The lines at the beginning and end of each message will contain the word or (EOF) that does not match the lowest level open structure. The precompiler assumes that the proper terminator for that nesting level was omitted. It therefore terminates the figure at that nest level and then examines the next figure within which the terminated figure was nested. This termination procedure continues until the nesting level is reached with the proper starting word. That figure is then closed and normal processing is resumed.

MSG 009 _________________ENCOUNTERED BEFORE CASE FIGURE TERMINATED. ENDCASE ASSUMED PRECEDING ____________.

This error message is generated whenever an intermediate or terminating structure word that is not part of the lowest level structure that is currently open has been detected. Before the message is selected for output it is first determined that a starting word does exist at some higher nesting level. An end of file condition (EOF) causes a similar action. The lines at the beginning and end of each message will contain the word or (EOF) that does not match the lowest level open structure. The precompiler assumes that the proper terminator for that nesting level was omitted. It therefore terminates the figure at that nest level and then examines the next figure within which the terminated figure was nested. This termination procedure continues until the nesting level is reached with the proper starting word. That figure is then closed and normal processing is resumed.

MSG 010 UNMATCHED ______ DELETED.

An intermediate or terminating structuring word for which no starting word of the same structure can be detected at any nested level has been ignored.

MSG 011 ERROR IN CASE NUMBER DETECTED.

A numeric value for a CASE was not detected. The word that was found is ignored by the program.

MSG 012 OPTION CARD ERROR.

The entire 80 column card is printed where the blank line is indicated.

MSG 014 END OF PROGRAM REACHED WITHIN DO FIGURE.

The end of input file was reached while attempting to process the DO figure.

MSG 015 END OF PROGRAM REACHED WITHIN IF FIGURE.

The end of input file was reached while attempting to process the IF figure.

MSG 016 END OF PROGRAM REACHED WITHIN CASENTRY FIGURE.

The end of input file was reached while attempting to process the CASE figure.

MSG 019 DUPLICATE CASE NUMBER FOUND.

Duplicate case numbers in a structured figure were detected. Second number discarded. Some following code may be unreachable.

H.7.2.2 Errors Which Terminate Processing

The error messages which follow are ones which suspend precompiler processing. Some of them are caused by exceeding the size of the various push down stacks used by the precompiler to store information. In some of these cases, if the program cannot be decreased in size it may be necessary to increase the size of the stack which overflowed and recompile the precompiler.

MSG 101 CASE NUMBER STACK OVERFLOWED. MAXIMUM OF 200 EXCEEDED. FATAL ERROR. EXECUTION TERMINATED.

This stack is currently set to hold a maximum of 200 CASE numbers. This is not meant to imply that only 200 CASE numbers may be present in a given program since once a CASE statement is terminated with an ENDCASE, the CASE numbers associated with it are removed from the stack.

MSG 102 NEST STACK OVERFLOWED. MAXIMUM OF 300 EXCEEDED. FATAL ERROR. EXECUTION TERMINATED.

This stack contains the starting and intermediate structure words of each structure set for which no terminating word has yet been encountered. The maximum depth to which figures may be stacked is set at 300.

MSG 103 NEST NUMBER STACK OVERFLOWED. MAXIMUM OF 50 EXCEEDED. FATAL ERROR. EXECUTION TERMINATED.

This stack contains the nest numbers of each structured figure which at any time in the program has not yet been terminated. The maximum depth to which figures may be nested is thus set at 50.

MSG 104 CONDITION STACK OVERFLOWED. MAXIMUM of 50 EXCEEDED. FATAL ERROR. EXECUTION TERMINATED.

The conditions on DO statements are saved in a stack and removed when the corresponding ENDDO is detected. The stack is set to hold 50 cards maximum.

MSG 105 CONDITION COUNT STACK OVERFLOWED. MAXIMUM OF 50 EXCEEDED. FATAL ERROR. EXECUTION TERMINATED.

This stack is also used with DO statements to keep track of the number of cards in each of the conditions stacked in the condition stack.

MSG 108 MAXIMUM NUMBER OF STRUCTURE FIGURES 999999 EXECUTED. FATAL ERROR. EXECUTION TERMINATED.

The digital value appended to all statement names generated by the precompiler to assure uniqueness has been exceeded.

APPENDIX I. GENERAL PREPROCESSOR

The General Preprocessor is an addition to the PSL system which provides the user an alternative method for processing unstructured source code. The General Preprocessor allows unstructured code two features that were previously only available to structured SCOBOL, SJOVIAL or SPFORT source code. These features are Data Compression and INCLUDE Processing.

The General Preprocessor obtains data compression by allowing unstructured source code to be stored in the SOURCE Section File in random blocks. Formerly, unstructured code was stored in independent files as sequential card images. The code in SOURCE Section Files has trailing blanks removed and with the CREATE function option COMPRESS=YES (which is the default for SOURCE sections), may have leading blanks removed also.

INCLUDE processing is handled upon compilation. When initiated by the COMPILE function, the General Preprocessor scans the unstructured source code for INCLUDE statements and inserts the included units. The format of the INCLUDE statement is described in section 3.2.8.

At present the General Preprocessor accommodates four languages: ASM, COBOL, FORTRAN and JOVIAL. To use the General Preprocessor the user need only ADD the unstructured source code with the keyword LANGUAGE set to <u>ASM</u>G, <u>COBOL</u>G, <u>FORTRAN</u>G or <u>JOVIAL</u>G. The unit type (keyword UTYPE) will default to main for these LANGUAGE keyword values. The General Preprocessor is called automatically by the COMPILE Function and is not requested by the user.

APPENDIX J. PSL SYSTEM MANAGEMENT FORMATS

The Management (MGMT) Section of the system project and library contains management data format units patterned after the management data collection requirements determined in Volume IX of the Structured Programming Series. An index of this MGMT Section or source listing of the referenced format units is given here to acquaint the PSL user with their contents. The formats may be moved (all or in part) to a user-specified MGMT Section (refer to paragraph 3.4.15 for specific details) after which they may be modified and/or directly utilized to facilitate the collection of management data.

```
PROJECT: FHACD146                DATE: 06/23/77     SECTION SIZE:    25 PSL BLOCKS                                   PAGE   1
LIBRARY: PSL                     TIME: 18:51                USED:    20 PSL BLOCKS
SECTION: MGMT                                              (12.8 PSL BLOCKS = 1 TRACK)
                                          INCLUDED
UNIT                             TYPE        COUNT   VER/MOD   UPDATED          UPDATED BY    LANGUAGE   LINES   CREATED
******************************   *************       *******   **************   ***********   ********   *****   ********

MDCR-FORMAT-JOB                  FORMAT              001/000   06/23/77 18:51   PSL                          7   06/23/77
MDCR-FORMAT-MODULE               FORMAT              001/000   06/23/77 18:51   PSL                         25   06/23/77
MDCR-FORMAT-SUBSYSTEM            FORMAT              001/000   06/23/77 18:50   PSL                         29   06/23/77
MDCR-FORMAT-SYSTEM               FORMAT              001/000   06/23/77 18:50   PSL                         80   06/23/77
MDCR-FORMAT-UNIT                 FORMAT              001/000   06/23/77 18:51   PSL                         22   06/23/77
MDCR-PLAN                        PLAN                000/000   06/23/77 18:50   PSL                          0   06/23/77
```

```
 1   01001      12PROJ-TITLE    PROJECT TITLE
 2   02001 RPT 48PROJ-DESCR    PROJECT DESCRIPTION
 3   03001     N06START-DATE   PROJECT START DATE
 4   04001     N06EST-END-DATE ESTIMATED COMPLETION DATE
 5   05001     N06ACT-END-DATE ACTUAL COMPLETION DATE
 6   06001     N03P-AVE-MGRS   PLANNED AVERAGE YEARS EXPERIENCE MANAGERS
 7   07001     N03P-AVE-ANAL   PLANNED AVERAGE YEARS EXPERIENCE ANALYSIS
 8   08001     N03P-AVE-PROG   PLANNED AVERAGE YEARS EXPERIENCE PROGRAMMERS
 9   09001     N03P-AVE-ADMIN  PLANNED AVERAGE YEARS EXPERIENCE ADMINISTRATIVE
10   10001     N03P-AVE-OTH    PLANNED AVERAGE YEARS EXPERIENCE OTHER
11   11001     N03A-AVE-MGRS   ACTUAL AVERAGE YEARS EXPERIENCE MANAGERS
12   12001     N03A-AVE-ANAL   ACTUAL AVERAGE YEARS EXPERIENCE ANALYSTS
13   13001     N03A-AVE-PROG   ACTUAL AVERAGE YEARS EXPERIENCE PROGRAMMERS
14   14001     N03A-AVE-ADMIN  ACTUAL AVERAGE YEARS EXPERIENCE ADMINISTRATIVE
15   15001     N03A-AVE-OTH    ACTUAL AVERAGE YEARS EXPERIENCE OTHER
16   16001     N04P-NBR MGRS   PLANNED NUMBER OF MANAGERS
17   17001     N04P-NBR-ANAL   PLANNED NUMBER OF ANALYSTS
18   18001     N04P-NBR-PROG   PLANNED NUMBER OF PROGRAMMERS
19   19001     N04P-NBR-ADMIN  PLANNED NUMBER OF ADMINISTRATIVE
20   20001     N04P-NBR-OTH    PLANNED NUMBER OF OTHER
21   21001     N04A-NBR-MGRS   ACTUAL NUMBER OF MANAGERS
22   22001     N04A-NBR-ANAL   ACTUAL NUMBER OF ANALYSTS
23   23001     N04A-NBR-PROG   ACTUAL NUMBER OF PROGRAMMERS
24   24001     N04A-NBR-ADMIN  ACTUAL NUMBER OF ADMINISTRATIVE
25   25001     N04A-NBR-OTH    ACTUAL NUMBER OF OTHER
26   26001     N03E-TURNOVER   ESTIMATED PERSONNEL TURNOVER RATE
27   27001     N03A-TURNOVER   ACTUAL PERSONNEL TURNOVER RATE
28   28001     N03E-LOC-TRIPS  ESTIMATED LOCAL TRAVEL
29   29001     N03A-LOC-TRIPS  ACTUAL LOCAL TRAVEL
30   30001     N03E-DIS-TRIPS  ESTIMATED DISTANT TRAVEL
31   31001     N03A-DIS-TRIPS  ACTUAL DISTANT TRAVEL
32   32001     N01E-WORD-COND  ESTIMATED WORKING CONDITIONS
33   33001     N01A-WORD-COND  ACTUAL WORKING CONDITIONS
34   34001     N02P-LANG-EXP   PLANNED PROGRAMMING LANGUAGE EXPERIENCE
35   35001     N02A-LANG-EXP   ACTUAL PROGRAMMING LANGUAGE EXPERIENCE
36   36001     N02P-SIM-EXP    PLANNED SIMILAR APPLICATION EXPERIENCE
37   37001     N02A-SIM-EXP    ACTUAL SIMILAR APPLICATION EXPERIENCE
38   38001     N02P-TARG-EXP   PLANNED TARGET COMPUTER EXPERIENCE
39   39001     N02A-TARG-EXP   ACTUAL TARGET COMPUTER EXPERIENCE
40   40001     N02E-APPL-EXP   Estimated CUSTOMER APPLICATION EXPERIENCE
```

```
41   41001     N02A-APPL-EXP   ACTUAL CUSTOMER APPLICATION EXPERIENCE
42   42001     N02E-EQUIP-EXP  ESTIMATED CUSTOMER EQUIPMENT EXPERIENCE
43   43001     N02A-EQUIP-EXP  ACTUAL CUSTOMER EQUIPMENT EXPERIENCE
44   44001     N01E-COMPLEXITY ESTIMATED COMPLEXITY OF PROJECT
45   45001     N01A-COMPLEXITY ACTUAL COMPLEXITY OF PROJECT
46   46001     N04EDOC-FUNC-SP ESTMATED PAGES DOCUMENTATION FUNCTIONAL SPECS
47   47001     N04EDOC-USER-GU ESTMATED PAGES DOCUMENTATION USERS GUIDE
48   48001     N04EDOC-TEST-SP ESTMATED PAGES DOCUMENTATION TEST SPECIFICATIONS
49   49001     N04EDOC-PROG-LT ESTIMATED PAGES DOCUMENTATION PROGRAM LISTINGS
50   50001     N04ADOC-FUNC-SP ACTUAL PAGES DOCUMENTATION FUNCTIONAL SPECS
51   51001     N04ADOC-USER-GU ACTUAL PAGES DOCUMENTATION USERS GUIDE
52   52001     N04ADOC-TEST-SP ACTUAL PAGES DOCUMENTATION TEST SPECIFICATIONS
53   53001     N04ADOC-PROG-LT ACTUAL PAGES DOCUMENTATION PROGRAM LISTINGS
54   54001      12SYS-NAME     SYSTEM NAME
55   55001  RPT 12SUBSYS-NAME  SUBSYSTEM NAMES
56   56001  RPTN04SUBSYS-LEVEL NUMBER SUBSYSTEM LEVELS
57   57001  RPTN04SUBSYS-PROG  NUMBER PROGRAMS FOR SUBSYSTEMS
58   58001     N04E-OVERLAP    ESTIMATED NUMBER OF NON-OVERLAPPING FIELDS
59   59001     N04A-OVERLAP    ACTUAL NUMBER OF NON-OVERLAPPING FIELDS
60   60001     N04E-INPUT-FOR  ESTIMATED NUMBER INPUT FORMATS
61   61001     N04A-INPUT-FOR  ACTUAL NUMBER INPUT FORMATS
62   62001     N03E-FLDS-IN    ESTIMATED AVERAGE FIELDS FOR INPUT FORMATS
63   63001     N03A-FLDS-IN    ACTUAL AVERAGE FIELDS FOR INPUT FORMATS
64   64001     N04E-OUTPUT-FOR ESTIMATED NUMBER OUTPUT FORMATS
65   65001     N04A-OUTPUT-FOR ACTUAL NUMBER OUTPUT FORMATS
66   66001     N03E-FLDS-OUT   ESTIMATED AVERAGE FIELDS FOR OUTPUT FORMATS
67   67001     N03A-FLDS-OUT   ACTUAL AVERAGE FIELDS FOR OUTPUT FORMATS
68   68001     N04P-SYS-TESTS  NUMBER SYSTEM TEST RUNS PLANNED
69   69001     N04S-SYS-TESTS  NUMBER SUCCESSFUL SYSTEM TEST RUNS
70   70001     N04X-SYS-TESTS  NUMBER SYSTEM TEST RUNS EXECUTED
71   71001      48DELETE-ITEM  ITEM TO BE DDELETED DURING TESTS
72   73001 S020TTLLINES-IN     LINES OF SOURCE CODE INPUT
73   74001 M130TTLCOMPILE-NBR  NUMBER OF COMPILATIONS
74   75001 S090TTLLINES-IN-CY  NUMBER OF LINES IN CYCLE
75   76001 S100TTLUPDATE-IN-CY NUMBER OF UPDATES IN CYCLE
76   77001 S110TTLNBR-UPDATES  NUMBER OF UPDATES
77   78001 S040TTLNBR-UNITS    NUMBER OF UNITS
78   79001 S050AVGAVE-U-SIZE   AVERAGE UNIT SIZE
79   80001 S060MAXMAX-U-SIZE   MAXIMUM UNIT SIZE
80   81001 S030TTLNBR-MODULES  NUMBER OF MODULES
```

```
 1   01001      12SUBSYS-NAME  SUBSYSTEM NAME
 2   02001 M000TTLLINES-IN     LINES OF SOURCE CODE INPUT
 3   03001 $001TTLNBR-MODULES  NUMBER OF MODULES
 4   04001 M070TTLNBR-UNITS    NUMBER OF UNITS
 5   05001 M080AVGAVE-U-SIZE   AVERAGE UNIT SIZE
 6   06001 M090MAXMAX-U-SIZE   MAXIMUM UNIT SIZE
 7   07001 M050AVGAVE-M-SIZE   AVERAGE MODULE SIZE
 8   08001 M060MAXMAX-M-SIZE   MAXIMUM MODULE SIZE
 9   09001 M140TTLLINES-IN-CY  NUMBER OF LINES IN CYCLE
10   10001 *110CYCUPDATE-IN-CY NUMBER OF UPDATES IN CYCLE
11   11001 M160TTLNBR-UPDATES  NUMBER OF UPDATES
12   12001 M170AVGAVE-UPDATES  AVERAGE NUMBER OF UPDATES
13   13001 M180MAXMAX-UPDATES  MAXIMUM NUMBER OF UPDATES
14   14001      06CURR-CY-DATE CURRENT CYCLE DATE
15   15001      06PREV-CY-DATE PREVIOUS CYCLE DATE
16   16001     N04MAN-MTH-BUD  MAN-MONTHS OF EFFORT BUDGET
17   17001     N04MAN-MTH-EXP  MAN-MONTHS OF EFFORT EXPENDED
18   18001     N08MAT-COST-BUD MATERIAL COSTS BUDGET
19   19001     N08MAT-COST-EXP MATERIAL COSTS EXPENDED
20   20001     N08PER-COST-BUD PERSONNEL COSTS BUDGET
21   21001     N08PER-COST-EXP PERSONNEL COSTS EXPENDED
22   22001     N08TRA-COST-BUD TRAVEL COSTS BUDGET
23   23001     N08TRA-COST-EXP TRAVEL COSTS EXPENDED
24   24001     N08COMP-TIME-BU COMPUTER TIME BUDGET
25   25001     N08COMP-TIME-EX COMPUTER TIME EXPENDED
26   26001     N08COMP-COST-BU COMPUTER COSTS BUDGET
27   27001     N08COMP-COST-EX COMPUTER COSTS EXPENDED
28   28001     N08MISC-COST-BU MISCELLANEOUS COST BUDGET
29   29001     N08MISC-COST-EX MISCELLANEOUS COST EXPENDED
```

```
 1   01001       12MODULE-NAME  MODULE NAME
 2   02001 A303   ORIG-PGMR     ORIGINATING PROGRAMMER
 3   03001 A302   ORIG-START    ORIGINATING START DATE
 4   04001 A305   LS-MOD-DATE   LAST MODIFIED DATE
 5   05001 A304   PROG-LANG     PROGRAM LANGUAGE
 6   06001 A101TTLLINES-IN      LINES OF SOURCE CODE INPUT
 7   07001 $001TTLNBR-UNITS     NUMBER OF UNITS
 8   08001 A101AVGAVE-U-SIZE    AVERAGE UNIT SIZE
 9   09001 A101MAXMAX-U-SIZE    MAXIMUM UNIT SIZE
10   10001 A100TTLLINES-ADDED   NUMBER OF LINES ADDED
11   11001 A10 TTLLINES-DELETE NUMBER OF LINES DELETED
12   12001 A10 TTLLINES-CHANGE NUMBER OF LINES CHANGE
13   13001 A20 TTLCOMPILE-NBR   NUMBER OF COMPILATIONS
14   14001 *060CYCLINES-IN-CY   NUMBER OF LINES IN CYCLE
15   15001 *170CYCAVER-UPDA-CY AVERAGE UPDATE CYCLE
16   16001 A107TTLNBR-UPDATES   NUMBER OF UPDATES
17   17001 A107AVGAVE-UPDATES   AVERAGE NUMBER OF UPDATES
18   18001 A107MAXMAX-UPDATES   MAXIMUM NUMBER OF UPDATES
19   19001       06CURR-CY-DATE CURRENT CYCLE DATE
20   20001       06PREV-CY-DATE PREVIOUS CYCLE DATE
21   21001 A106TTLLINES-COPIED LINES OF SOURCE CODE COPIED
22   22001 A202   REAL-UNIT-CT REAL UNIT COUNT
23   23001 A203   STUB-UNIT-CT STUB UNIT COUNT
24   24001 A204   SP-ERROR-CT  STRUCTURED PROGRAM ERROR COUNT
25   25001 A301   UNIT-TYPE    MODULE TYPE
```

```
1   01001       12PROG-ID       PROGRAM IDENTIFICATION
2   02001  RPT 12JOB-NAME       JOB NAME
3   04001      N04NBR-RUNS      NUMBER OF RUNS
4   05001      N04NBR-RUNS-CY   NUMBER OF RUNS FOR CYCLE
5   06001       06CURR-CY-DATE  CURRENT CYCLE DATE
6   07001       06PREV-CY-DATE  PREVIOUS CYCLE DATE
7   08001  RPTN03AVE-RUN-JOB    AVERAGE TURNAROUND TIME PER JOB
```

```
 1   01001 A001   UNIT-TYPE    UNIT TYPE
 2   02001 A002   LINE-SIZE    LINE SIZE
 3   03001 A003   INCL-NAME    INCLUDE UNIT NAME
 4   04001 A004   VERSION      VERSION
 5   05001 A005   MODIFICATION MODIFICATION
 6   06001 A006   DATE-ORGIN   DATE ORGINATED
 7   07001 A007   ORIGINATOR   ORIGINATORS NAME
 8   08001 A008   LAST-UPDATE  DATE OF LAST UPDATE
 9   09001 A009   TIME-UPDATE  TIME OF LAST UPDATE
10   10001 A010   LANGUAGE     UNIT LANGUAGE
11   11001 A011   INCL-COUNT   INCLUDE COUNT
12   12001 A012   LINES-IN-UNI LINES IN UNIT
13   13001 A013   SP-ERROR-SW  STRUCTURED PROGRAM ERROR SWITCH
14   14001 A014   LINES-ADDED  LINES ADDED TO UNIT
15   15001 A015   LINES-CHANGE LINES CHANGED IN UNIT
16   16001 A016   LINES-DELETE LINES DELETED IN UNIT
17   17001 A017   LINES-IN-TOT LINES INPUT TOTAL
18   18001 A018   LINES-IN-CYC LINES INPUT CYCLE
19   19001 A019   LINES-COPIED LINES COPIED IN UNIT
20   20001 A020   UPDATES-TOT  UPDATES TOTAL
21   21001 A021   UPDATES-CYC  UPDATES CYCLE
22   22001 A022   USER-WORK    USER WORK AREA
```

END

DATE

FILMED

12-81

DTIC